



UNIVERSIDAD
DE MÁLAGA



ESCUELA DE INGENIERÍAS INDUSTRIALES

Ingeniería de Sistemas y Automática

Ingeniería de Sistemas y Automática

TRABAJO FIN DE GRADO

**MODELADO Y SIMULACIÓN DE DRONES CON PIXHAWK-4
CON ROS Y GAZEBO**

Grado en

Ingeniería Electrónica, Robótica y Mecatrónica

Autor: ÁLVARO BORGES SUÁREZ

Tutor: ALFONSO JOSÉ GARCÍA CEREZO

Cotutor DAHUI LIN YANG

MÁLAGA, Septiembre de 2.021



UNIVERSIDAD
DE MÁLAGA



DECLARACIÓN DE ORIGINALIDAD DEL PROYECTO/TRABAJO FIN DE GRADO

D./ Dña.: Álvaro Borges Suárez

DNI/Pasaporte: 78648466N Correo electrónico: alvaritoborges99@gmail.com

Titulación: Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Título del Proyecto/Trabajo: Modelado y Simulación de Drones con PIXHAWK-4 con ROS y GAZEBO.

DECLARA BAJO SU RESPONSABILIDAD

Ser autor/a del texto entregado y que no ha sido presentado con anterioridad, ni total ni parcialmente, para superar materias previamente cursadas en esta u otras titulaciones de la Universidad de Málaga o cualquier otra institución de educación superior u otro tipo de fin.

Así mismo, declara no haber trasgredido ninguna norma universitaria con respecto al plagio ni a las leyes establecidas que protegen la propiedad intelectual, así como que las fuentes utilizadas han sido citadas adecuadamente.

En Málaga, a 14 de agosto de 2021.

Fdo.:

Resumen

El desarrollo de un software de control para sistemas robotizados, puede ser una tarea compleja. Por ello se hace indispensable el uso de simuladores y el desarrollo de modelos que permitan aproximarse al comportamiento del sistema.

Este proyecto se enfocará en el estudio y familiarización de un entorno de simulación de drones y en el desarrollo de una interfaz de comunicación entre ROS y el UAV. Finalmente se elaborará un modelado de primer orden del dron simulado, sirviendo como guía para el posterior modelado de un dron del departamento de ingeniería de sistemas y automática.

Palabras Claves: UAV, ROS, Gazebo, Ardupilot, SITL, GCS, MAVLink, MAVROS, modelo.

Índice

1.	Introducción.....	1
2.	Elementos del Proyecto.....	5
2.1	ROS.....	5
2.1.1	Introducción.....	5
2.1.2	Nodos	5
2.1.3	Comunicación	6
2.1.4	Paquetes	7
2.1.5	Comandos	7
2.2	PIXHAWK-4	8
2.3	Ardupilot.....	9
2.3.1	SITL	9
2.4	Gazebo	9
2.5	MAVLink.....	10
2.6	MAVROS.....	10
2.7	Estación de control terrestre (GCS).....	10
2.7.1	MAVProxy	10
2.7.2	QGroundControl	11
2.8	Modelado.....	11
2.8.1	Sistemas de primer orden	12
2.8.2	Ecuaciones cinemáticas de primer orden de un dron (variables de estado)....	13
3.	Instalaciones y configuraciones	15
3.1	Creación de una partición e instalación de Ubuntu	15
3.1.1	Creación de partición.....	15
3.1.2	Generación USB con Ubuntu	15
3.1.3	Instalación de Ubuntu	16
3.2	Instalación y configuración de ROS.....	17
3.2.1	Instalación.....	17
3.2.2	Configuración del Entorno.....	17
3.3	Instalación del simulador SITL de ardupilot y el plugin de Gazebo	18
3.4	Instalación del paquete MAVROS	19
3.5	Instalación de QGroundControl	19
4.	Implementación.....	20
4.1	Inicio del entorno de simulación.....	20
4.1.1	Prueba del simulador con MAVProxy	21

4.2	Estudio de QGroundControl	26
4.2.1	Despegue	26
4.2.2	Movimiento simple a un punto.....	27
4.2.3	Misiones en QGroundControl	28
4.3	MAVROS.....	32
4.4	Elaboración de la interfaz.....	37
4.4.1	Copter_node	37
4.4.2	Mission_node	44
4.5	Modelado.....	49
4.5.1	Toma de datos	49
4.5.2	Estimación del modelo en Matlab	50
4.5.3	Comparación de funciones estimadas con valores reales.....	53
5.	Conclusión.....	56
6.	Trabajos Futuros	56
7.	Contenido del CD	56
	Bibliografía.....	57
	Anexos.....	59
A.	Códigos.....	59
A.1	copter_node.py	59
A.2	mission_node.py	62
A.3	Prueba_wp.py	64
A.4	Modelado.py	66
A.5	medidas.m.....	68
B.	Modificación de CMakeList.....	69

1. Introducción

Los avances tecnológicos de los últimos años en el campo de la automatización de vehículos aéreos, han contribuido a la aparición de los UAV (*Unmanned aerial vehicle* o vehículo aéreo no tripulado), más conocidos con el término dron. Estos se tratan de vehículos aéreos capaces de volar sin la necesidad de un tripulante. Su historia se remonta a principios del siglo XX con la creación del primer piloto automático por Sperry Corporación [1]. Este dispositivo buscaba aliviar la presión de los pilotos en pleno vuelo al estabilizar automáticamente la aeronave haciendo uso de giróscopos. Más adelante, en el transcurso de la primera guerra mundial, se utilizaría esta tecnología para el desarrollo del avión *hewitt-Sperry* (Figura 1), un proyecto que pretendía emplear aeronaves no tripuladas cargadas de explosivos como bombas aéreas.



Figura 1: Avión automático hewitt-Sperry.

En los años 30 se empezaron a desarrollar las primeras aeronaves controladas mediante ondas de radio. Posteriormente, aparece el primer misil guiado, la V-1 (*Vergeltungswaffe 1* o arma de represalia 1) desarrollada por la *Luftwaffe* (la fuerza aérea alemana) durante la segunda guerra mundial. Al finalizar la guerra, Estados Unidos desarrollo varias aeronaves no tripuladas para su uso como blancos en prácticas de vuelo [2]. Destacando el modelo Q2 de la marca *Ryan Aeronautical company*, el cual se ha convertido en el predecesor de la famosa serie “Firebee”. El uso de los drones empieza a cobrar más importancia durante la guerra de Vietnam, donde se utilizan modelos más sofisticados como el AQM-34 (Figura 2) para reconocimiento de las líneas enemigas.



Figura 2: Dron militar AQM-34 Firebee durante la guerra de Vietnam.

Es innegable el pasado bélico que acompaña a este tipo de vehículos, sin embargo, hoy en día su uso no se restringe al terreno militar, encontrando aplicaciones en numerosos campos como:

- Agricultura: fumigación, estudios de la fertilidad de los terrenos y estado de los cultivos mediante cámaras multiespectrales.
- Monitorización ambiental y estudios geofísicos como labores de cartografía.
- Labores de búsqueda y rescate de víctimas de catástrofes.
- Monitorización de lugares poco accesibles.
- Entrega de paquetes (*Figura 3*).



Figura 3: Dron de la empresa UPS entregando un paquete.

Los UAV comúnmente se dividen en tres categorías[3]:

- Ala fija (*Figura 4*): son los que más resistencia y autonomía ofrecen. No obstante, son poco maniobrables, debido a que no pueden girar sobre su propio eje ni despegar ni aterrizar en vertical, lo que obliga a disponer de una pista de aterrizaje lo bastante amplia como se requiera.



Figura 4: Dron de ala fija militar MQ-9 Reaper.

- Multirrotores (*Figura 5*): son el tipo de dron más conocido y utilizado tanto en entornos profesionales como de ocio. Ello se debe en parte a la simplicidad de su mecanismo, además de que ofrece una gran estabilidad y maniobrabilidad. Este tipo de drones comúnmente se dividen en otras categorías según el número de rotores de los que dispone (cuadricóptero, hexacóptero, etc.). La adición de motores aporta

mayor estabilidad, sin embargo, disminuye su autonomía a la vez que su capacidad de carga.



Figura 5: Cuadricóptero VOXL M500.

- Helicópteros (*Figura 6*): ofrecen una mayor autonomía y resistencia que los sistemas multirrotores, además de poseer una gran capacidad de carga, pudiendo transportar objetos con un peso de varios kilogramos. Su gran desventaja es lo complicado que resulta pilotarlos además de que necesitan diariamente de ajustes para poder volar en las condiciones más óptimas.



Figura 6: Dron helicóptero de reconocimiento ruso BVS VT 500.

Los drones están conformados por cuatro partes básicas: un sistema de propulsión que eleve al dron y lo mantenga en el aire, un soporte o chasis que contenga todos los elementos del UAV, un sistema de alimentación mediante baterías y un autopiloto. Este último se trata de un controlador de vuelo encargado de gobernar al dron para que pueda seguir un determinado plan u orden. Para que los drones puedan funcionar de manera autónoma es indispensable adquirir multitud de datos tanto internos como externos a la propia aeronave, los cuales se obtienen mediante la adición de múltiples sensores. Podemos dividir los sensores en dos categorías:

- Sensores Propioceptivos: permiten conocer el estado actual del dron como su posición y orientación, ejemplo de esto son el GPS y las unidades de medición inercial (IMU).
- Sensores Exteroceptivos: permiten al dron conocer el mundo que le rodea, ejemplo de estos son las cámaras y los Lidar.

Un UAV puede ser controlado tanto por un piloto a distancia como por un software de control. El principal beneficio del desarrollo de un software de control, es la automatización de procesos, de forma que el dron sea capaz de realizar tareas sin la necesidad de contar con un piloto (resultando en un sistema más eficiente). No obstante, el desarrollo de este software

representa una cierta complejidad, ya que debe adaptarse a las características de cada dron. Siendo indispensable la elaboración con anterioridad de un modelo que represente de la manera más fiel posible el comportamiento del sistema real. También son de vital importancia en el desarrollo de este tipo de software, el uso de simuladores debido a que permiten estudiar el funcionamiento de dichos códigos sin la necesidad de correr los posibles riesgos derivados de un sistema real.

El objetivo de este proyecto es el estudio y familiarización de un entorno de simulación de drones, así como el desarrollo de una interfaz de comunicación entre dicho simulador y un UAV usando ROS y Gazebo, y el posterior modelado de un dron del departamento de ingeniería de sistemas y automática.

2. Elementos del Proyecto

2.1 ROS

2.1.1 Introducción

ROS [4] (*Robot Operating System*) es un entorno de trabajo o conjunto de herramientas, librerías y convenciones, que facilita el desarrollo de software orientado a sistemas robóticos. ROS inicialmente fue creado con el objetivo de fomentar el trabajo en grupo, de tal forma que múltiples entidades puedan compartir sus progresos en diferentes campos como el mapeo de zonas, la navegación autónoma o la localización desarrollando así de manera conjunta el software de control de un robot. Ello, unido con la gran comunidad que posee actualmente y el hecho de que es un proyecto *Open Source* (accesible y modificable por cualquier persona) hacen de ROS una buena opción para el desarrollo de proyectos centrados en la robótica.

Desde su lanzamiento en el año 2007, ROS cuenta con multitud de versiones. La más actual es *Noetic Ninjemys*, sin embargo, para el desarrollo del proyecto se ha elegido la versión anterior *Melodic Morenia* (Figura 7), lanzada el 23 de mayo de 2018, debido a que es la más recomendable por el grupo de Ardupilot para el uso del simulador.



Figura 7: Logotipo de ROS Melodic.

Actualmente ROS no es multiplataforma, orientándose únicamente para Ubuntu (Linux) y aunque se están desarrollando distribuciones para sistemas como Debian, Fedora e incluso Windows, estas versiones todavía no son del todo estables. Inicialmente se consideró el uso de una máquina virtual, sin embargo, debido a la carga computacional que representa el simulador, finalmente se ha optado por la creación de una partición para el uso conjunto de Windows y Ubuntu en el mismo ordenador.

2.1.2 Nodos

El funcionamiento de ROS es modular, donde a cada programa se le conoce como nodo. Cada nodo puede ejercer una determinada tarea, teniendo por ejemplo uno que se encargue de la comunicación con la controladora de hardware, otro de la elaboración de trayectorias, etc.

De entre todos los nodos existe uno al cual se le conoce como “Master”. Este se encarga de registrar los distintos nodos, topics y servicios que estén disponibles o se encuentren en

ejecución, así como de establecer las comunicaciones entre nodos, siendo, por lo tanto, el primero que se debe ejecutar.

Los nodos pueden escribirse tanto en lenguaje C++ como en Python mediante el uso de las dependencias roscpp y rospy, respectivamente. La funcionalidad de un nodo no depende del lenguaje en el que esté implementado, siendo posible la comunicación entre nodos definidos en distintos lenguajes. Para este proyecto se ha decidido utilizar Python, debido a que su sencillez y claridad lo han transformado en uno de los lenguajes más utilizados en los últimos años, siendo conocido su uso en campos como el de la inteligencia artificial o *Data Science*.

2.1.3 Comunicación

Los nodos pueden comunicarse entre sí mediante tres formas distintas: topics, servicios y/o parámetros.

Los topics implementan una comunicación asíncrona en la que un nodo puede publicar/subscribe a uno o varios topics. En la *Figura 8* se muestra un ejemplo de este tipo de comunicación. El nodo talker publica mensajes en el topic chatter, los cuales llegan al nodo listener debido a que está suscrito a dicho topic. Este tipo de comunicación es unidireccional lo que significa que cuando un nodo publica un mensaje en un topic, no tiene forma de saber si el mensaje ha llegado correctamente al destino, o incluso, si hay algún nodo que esté suscrito a dicho topic.



Figura 8: Ejemplo de comunicación mediante topics.

Los servicios, utilizan una comunicación del tipo síncrona en la que existen dos tipos distintos de nodos: los clientes y los servidores. Los clientes realizan una petición a los servidores. Estos ejecutan una serie de instrucciones, devolviendo una respuesta al cliente (al contrario que en la comunicación mediante topics) (*Figura 9*).



Figura 9: Esquema de comunicación mediante servicios.

El servidor de parámetros [5] es un diccionario compartido de variables, donde cada nodo puede allí almacenar y leer datos en tiempo de ejecución. Este servidor es útil para almacenar parámetros de configuración del sistema que necesiten ser consultados desde varios nodos.

2.1.4 Paquetes

Los paquetes son un conjunto organizado de archivos como nodos, definiciones de mensajes y servicios, modelos, etc. Para ser considerados paquetes deben de cumplir una serie de requisitos:

- Deben contener un archivo llamado “package.xml” que defina propiedades como el nombre del paquete, de los autores o el número de versión y otro denominado “CMakeLists.txt” que defina las dependencias que se van a utilizar y los archivos que se deben de compilar.
- Cada paquete debe tener su propio directorio.

Los paquetes son una manera útil y sencilla para compartir código. En la red pueden encontrarse múltiples paquetes gratuitos elaborados por la comunidad que resuelven varios de los problemas típicos en el desarrollo de sistemas robóticos.

2.1.5 Comandos

ROS posee una serie de comandos que serán de utilidad para el desarrollo del proyecto:

- `roslaunch`: se utiliza para ejecutar un nodo. Está formado por dos campos, el paquete donde se encuentra el nodo y el propio nodo en cuestión.

```
roslaunch <Paquete> <nodo>
```

- `roslaunch`: se utiliza para ejecutar varios nodos a la vez y especificar algunos valores iniciales en el servidor de parámetros. Para ello primero se deberá definir un fichero “.launch” en formato XML que especifique los nodos que se deben iniciar y los parámetros a modificar. Este comando se puede utilizar sin haber lanzado antes el nodo master, ya que lo lanzara automáticamente si no estuviese iniciado.

```
roslaunch <Paquete> <fichero.launch>
```

- `roslaunch`: permite trabajar con archivos de extensión “bag”, un formato estándar de ROS para el almacenamiento de datos. Para grabar los contenidos de ciertos topics se debe utilizar el argumento `record` tal y como se muestra a continuación:

```
roslaunch record <topic 1> <topic 2> ...
```

- `roslaunch`: muestra información relacionada sobre los nodos en ejecución. Para este proyecto se consideran interesantes los siguientes argumentos:
 - `list`: muestra la lista de nodos activos.
 - `info`: muestra la lista de topics donde publica y está suscrito un determinado nodo, así como la lista de servicios que ofrece.

```
roslaunch list
```

```
roslaunch info <nodo>
```

- `roslaunch`: muestra información relacionada sobre los topics activos. En este proyecto se hará uso de los siguientes argumentos:

- list: muestra la lista de topics activos.
- info: indica el tipo de mensaje de un determinado topic, así como la lista de nodos que publican y que están suscritos a dicho topic.
- pub: permite publicar un mensaje en un determinado topic.
- echo: muestra por pantalla los mensajes que llegan a un topic.

```
rostopic list
rostopic info <topic>
rostopic pub <topic> <tipo de mensaje> <mensaje>
rostopic echo <topic>
```

- roservice: muestra información relacionada sobre los servicios disponibles. Se destacan los subcomandos:
 - list: indica la lista de servicios disponibles.
 - call: Realiza la llamada a un determinado servicio pasándole los argumentos correspondientes.
 - info: muestra el nodo que ofrece el servicio, el tipo y los argumentos requeridos.

```
rosservice list
rosservice call <servicio> <lista de argumentos>
rosservice info <servicio>
```

2.2 PIXHAWK-4

PixHawk 4 [6] es un microcontrolador de vuelo diseñado y elaborado por *Holybro* en colaboración con *Auterion* y es utilizado por el dron del departamento que más adelante se pretende modelar. Este dispositivo cuenta con dos IMU distintas, un barómetro y un Magnetómetro integrados, así como varios puertos para Telemetría, uso de GPS, alimentación, y control de motores mediante PWM, tal y como se observa en la *Figura 10*. Normalmente viene preinstalado el software de control PX4, sin embargo, para este proyecto se usará Ardupilot ya que es el que ha instalado el fabricante del dron.

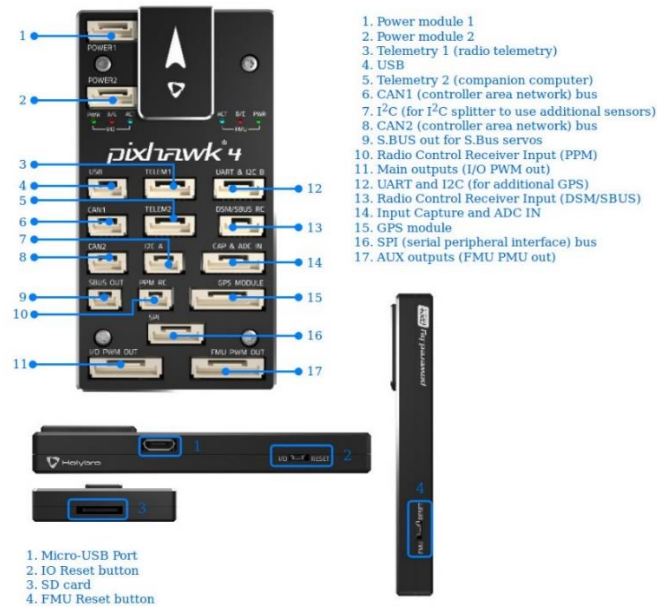


Figura 10: Esquema de conexiones del controlador PixHawk-4.

2.3 Ardupilot

Ardupilot [7] es una plataforma software de código abierto de piloto automático fiable y versátil, siendo capaz de instalarse y utilizarse en multitud de tipos de vehículos. Su desarrollo comenzó en el año 2009 y en el proceso han participado profesionales de distintos ámbitos. El hecho de que sea un proyecto *Open Source*, y de que disponga de una gran comunidad de desarrolladores, provoca que esté siempre en continua evolución a la par con el desarrollo tecnológico. Es por ello que está siendo utilizado por grandes corporaciones como la NASA e Intel, además de, multitud de universidades de todo el mundo.

2.3.1 SITL

Software in the Loop, abreviado como SITL [8], es un simulador que permite ejecutar Ardupilot en el ordenador sin necesidad de ningún tipo de hardware externo (como podría ser un dron). Entre los sistemas que se pueden simular encontramos vehículos terrestres, submarinos y drones (tanto de ala fija como multirrotores), además de una gran variedad de sensores. Una de las características que posee este simulador es que puede interactuar con otros externos, como Gazebo.

2.4 Gazebo

Gazebo [9] (Figura 11) es un entorno de simulación de robótica 3D de código abierto. Entre sus virtudes se encuentran los motores de físicas con los que trabaja, como ODE y Bullet; un potente motor de gráficos y una gran cantidad de modelos de robots comerciales, así como la posibilidad de simular múltiples tipos de sensores, desde Lidar hasta cámaras. Es importante destacar la buena interacción que tiene este simulador con ROS, lo cual es uno de los grandes

motivos que ha catapultado su popularidad hasta tal grado que es posible incluso su instalación conjunta.

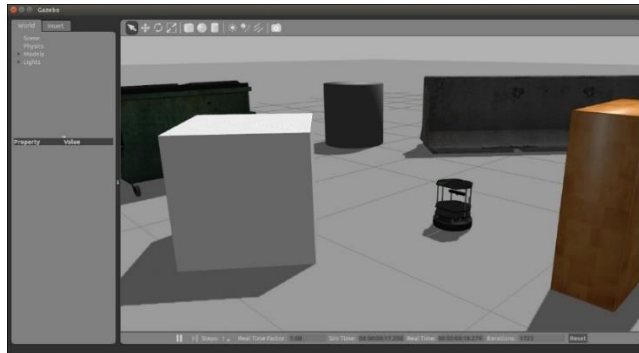


Figura 11: Simulación de turtlebot en Gazebo.

2.5 MAVLink

MAVLink [10] (*Micro Air Vehicle Link*) es un protocolo de comunicaciones usado para el intercambio de datos en sistemas no tripulados. Se caracteriza por ser eficiente y fiable dando medidas para detectar paquetes descartados y corruptos, además de añadir la autenticación de paquetes. El método de comunicación es un híbrido entre publicación/suscripción y diseño punto a punto, permitiendo la conexión de hasta 255 dispositivos. Estas conexiones pueden ser tanto externas (UAV y estación de control de tierra) como internas (UAV y cámara).

2.6 MAVROS

Dado que en el proyecto se pretende utilizar ROS para el control del UAV y el posterior modelado, se hace necesario implementar una interfaz que comunique ROS con el dron. Para ello se hará uso de MAVROS [11], un paquete de ROS que es capaz de convertir mensajes del protocolo MAVLink a topics de ROS y viceversa, permitiendo la comunicación entre ROS, el dron y la estación de control de tierra.

2.7 Estación de control terrestre (GCS)

Los GCS, (*Ground Control Station* o estaciones de control de tierra), son sistemas que permiten la monitorización y el control de drones. Estos se conforman, principalmente por uno o varios ordenadores conectados físicamente al dron mediante antenas y una aplicación software desde la cual se puede comandar y monitorizar la aeronave. Existen varias de estas aplicaciones, de las cuales en este proyecto se usarán dos de ellas, MAVProxy y QGroundControl.

2.7.1 MAVProxy

MAVProxy [12] es un software GCS, que es capaz de comunicarse con cualquier dispositivo que utilice el protocolo MAVLINK. Ofrece un control mediante línea de comandos y puede complementarse fácilmente con otras estaciones de control terrestres (como

QGroundcontrol). Además, puede instalarse y utilizarse en cualquier sistema operativo que cumpla con el estándar POSIX, como Linux, Windows y OS X.

2.7.2 QGroundControl

QGroundcontrol [13] (Figura 12) es un software GCS gratuito, pensado para vehículos que ejecuten PX4 y Ardupilot (o cualquier otro autopiloto que pueda comunicarse mediante MAVLink). Es una aplicación sencilla e ideal para aquellos que se inicien en el mundo de los UAV, donde el programa se conforma por una interfaz gráfica con una serie de botones que permiten de manera fácil mandar comandos al dron como pueden ser el despegue, el aterrizaje y la parada; o la elaboración de planes de vuelo. Otra característica que lo hace interesante es que es multiplataforma pudiendo instalarse en sistemas con Windows, OS X, distribuciones Linux y dispositivos móviles IOS y Android.



Figura 12: Icono de QGroundControl.

En este proyecto se realizará un estudio del funcionamiento de este software, para más adelante hacer uso de éste en paralelo con ROS, aumentando así la robustez del sistema de control, de tal forma que, si por alguna razón ROS fallase o la comunicación se interrumpiese, todavía podría seguir comandándose el UAV mediante QGroundControl.

2.8 Modelado

Un modelo es una representación abstracta de un sistema complejo que ayuda a comprender su funcionamiento al aproximarse a su comportamiento real ante determinadas entradas. Ello hace que sean de vital importancia para el desarrollo del software de control en sistemas autónomos. En este proyecto se elaborará un modelo sobre el dron del simulador, sirviendo de esquema para el posterior modelado de un dron real. Hay que recalcar que el modelo que se realizará será de alto nivel (modelo cinemático), es decir, que con este no se pretende estabilizar el dron (ya que eso ya lo realiza Ardupilot) sino automatizarlo. La estrategia que se ha seguido para modelar el dron, es considerar que cada uno de los ejes de movimiento (XYZ) y el giro en el eje Z (guiñada) no tienen ninguna relación entre sí y además que sus comportamientos son los de un sistema de primer orden. En la práctica esto no es del todo cierto, ya que mediante este modelo no se están considerando efectos de orden superior ni el acoplamiento de los movimientos mencionados previamente. No obstante, este tipo de modelo es bastante potente ya que es sencillo de obtener y funciona para la mayor parte de aplicaciones.

2.8.1 Sistemas de primer orden

Un sistema de primer orden es aquel que está representado mediante ecuaciones diferenciales de primer orden (primera derivada). Este tipo de sistemas usualmente poseen una función de transferencia como la que se expresa en la *Ecuación 1*.

$$G(s) = \frac{K}{\tau s + 1}$$

Ecuación 1: Función de transferencia modelo de primer orden.

Donde los parámetros que definen el comportamiento del sistema son:

- Ganancia (K): relación entre la entrada y la salida del sistema en régimen permanente.
- Constante de tiempo (τ): es el tiempo que tarda el sistema en alcanzar el 63,2 % de su valor final. Se suele relacionar con la velocidad de respuesta del sistema. A partir de este parámetro también se puede calcular el tiempo que tarda la señal en alcanzar el valor final (tiempo de establecimiento) utilizando la *Ecuación 2*.

$$T_s = 4\tau$$

Ecuación 2: tiempo de establecimiento (error del 2%).

En la Figura 13 se representa el tipo de respuesta que otorga este tipo de sistemas ante una entrada escalón.

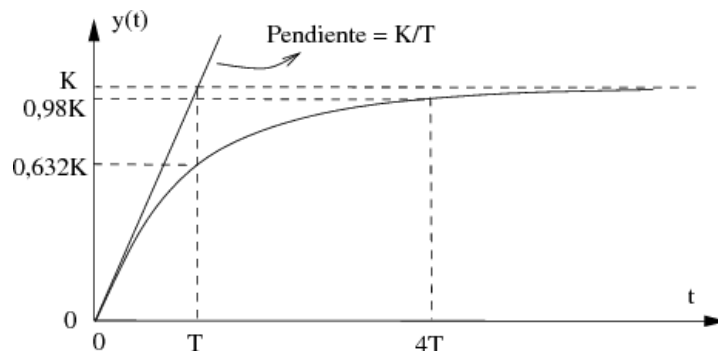


Figura 13: Respuesta de un sistema de primer orden ante entrada escalón unitario.

En ocasiones los sistemas reales sufren retardos a la hora de presentar una respuesta ante una determinada entrada. La *Ecuación 3* muestra la función de transferencia genérica de un modelo de primer orden después de añadir el efecto del retardo, donde ϑ es el tiempo de demora en segundos.

$$G(s) = \frac{K}{\tau s + 1} e^{-\vartheta s}$$

Ecuación 3: Función de transferencia modelo de primer orden con retardo.

En la *Figura 14* se puede comprobar el funcionamiento de este tipo de sistemas ante una entrada escalón. La respuesta obtenida es igual a la del caso anterior salvo en el desplazamiento de ϑ segundos. Por lo que el tiempo de establecimiento de este tipo de sistemas puede calcularse añadiendo a la *Ecuación 2* el tiempo de retardo (*Ecuación 4*).

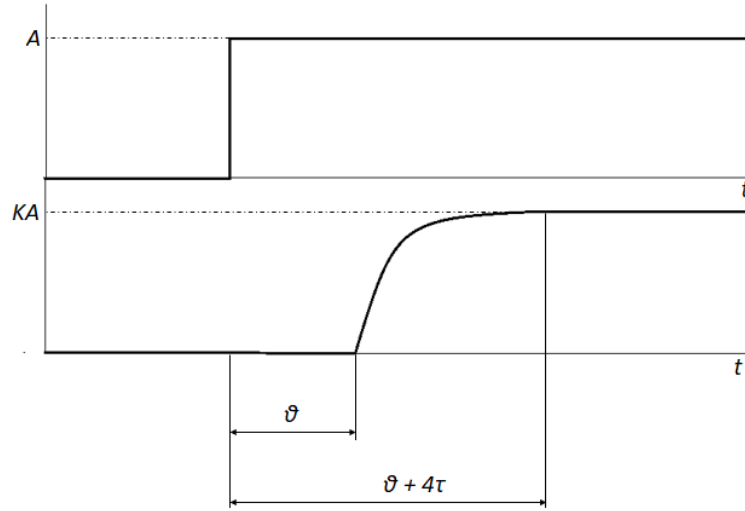


Figura 14: Respuesta de un sistema de primer orden con retardo ante entrada escalón.

$$T_s = \vartheta + 4\tau$$

Ecuación 4: Tiempo de establecimiento modelo con retardo (error 2%).

2.8.2 Ecuaciones cinemáticas de primer orden de un dron (variables de estado)

Teniendo en cuenta las consideraciones anteriores, la entrada que recibirá el dron será la siguiente $u = [u_x \ u_y \ u_z \ u_\gamma]^T$, que corresponde a las velocidades lineales XYZ y de giro en Z con respecto al sistema de coordenadas local a la aeronave. Por otro lado, el modelo cinemático del dron estará representado mediante las siguientes ecuaciones (*Ecuación 5*) [14]:

$$\begin{aligned} \dot{P} &= R(\gamma)v \\ \dot{v}_i &= \frac{1}{\tau_i}(-v_i + k_i u_i), \quad i \in \{x_1 y_1 z\} \\ \dot{v}_\gamma &= \frac{1}{\tau_\gamma}(-v_\gamma + k_\gamma u_\gamma) \end{aligned}$$

Ecuación 5: Modelo cinemático del dron.

Donde $P = [x \ y \ z]$ es el vector de posición del UAV con respecto al origen de coordenadas, $R(\gamma)$ la matriz de rotación con respecto al ángulo de guiñada (*Ecuación 6*) y $v = [v_x \ v_y \ v_z]^T$ el vector de velocidad de salida en el sistema de coordenadas local a la aeronave.

$$R(\gamma) = \begin{pmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Ecuación 6: Matriz de rotación con respecto al ángulo de guiñada.

En cuanto a los parámetros k y τ , son las respectivas constantes de primer orden para cada uno de los ejes de velocidad lineal y de giro en el eje Z. Por tanto, para obtener el modelo del dron será necesario obtener la salida real ante determinadas entradas y estimar este comportamiento a funciones de primer orden.

Finalmente, el estado del dron quedará definido mediante las siguientes variables (*Ecuación 7*):

$$X = [P \ v \ \gamma \ \dot{\gamma}]^T$$

Ecuación 7: Variables de estado del dron.

3. Instalaciones y configuraciones

3.1 Creación de una partición e instalación de Ubuntu

Para la instalación conjunta de Ubuntu junto a Windows se debe realizar una partición del disco duro para hacer espacio al nuevo sistema operativo, seguidamente preparar una unidad USB con la distribución elegida y por último arrancar el ordenador desde el USB.

3.1.1 Creación de partición

Para la creación de la partición del disco se deben seguir los siguientes pasos:

- Desde el menú de inicio ejecutar el administrador de discos.
- Una vez allí, clic derecho en la unidad C: y escoger la opción “reducir volumen”.
- Saldrá un menú donde se pide el tamaño en MB de la nueva partición. Para este proyecto se ha elegido aproximadamente 100 GB.

Si todo se ha realizado correctamente, se obtiene un resultado como el de la *Figura 15*, donde el espacio no asignado en negro corresponde a la partición creada.

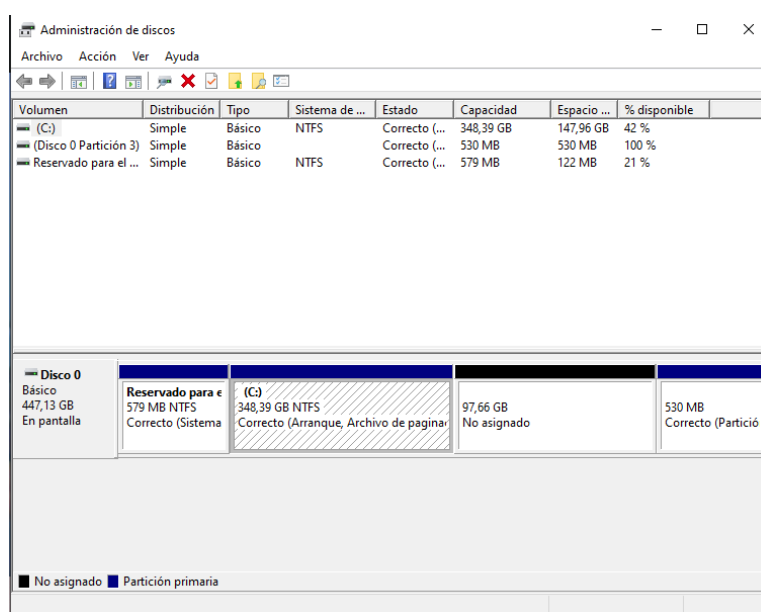


Figura 15: Administrador de discos con partición creada.

3.1.2 Generación USB con Ubuntu

Una vez creada la partición se procede a preparar el USB con Ubuntu, para ello se debe disponer de una unidad USB de mínimo 8 GB y de la imagen ISO del sistema a instalar. Para este proyecto se ha elegido la versión 18.04.5, también conocida como *Bionic Beaver* (castor biónico) [15], dado que más tarde se instalará la distribución *Melodic* de ROS.

Para introducir Ubuntu en el USB se ha hecho uso del programa Balena Etcher. Los pasos a seguir son:

- Con el programa Balena Etcher iniciado, elegir la opción “Flash from file” y escoger la ISO del sistema operativo.
- Con el USB conectado, pulsar “Select target” y elegir la unidad USB.
- Pulsar el botón “Flash!”.

3.1.3 Instalación de Ubuntu

El siguiente paso es reiniciar el equipo y arrancarlo desde el USB, para ello se ha tenido que cambiar el método de arranque desde la BIOS. Una vez arranque, aparecerá un menú de inicio. Para comenzar con la instalación debe escogerse la opción “Instalar Ubuntu” y seguir las siguientes indicaciones:

- Seleccionar el idioma, la distribución del teclado y configurar la red WiFi.
- Elegir la opción “Instalación normal”.
- Una vez llegado a la pestaña “Tipo de instalación”, escoger “Más opciones”.
- Aparecerá una ventana de gestión de particiones. Para añadir particiones nuevas debe pulsarse el botón con el símbolo “+”.
- Hay varias formas de configurar Ubuntu dependiendo de que particiones se añadan y su tamaño. Para la realización de este proceso se ha seguido un tutorial del medio Xataka[16] en el que especifican tres particiones básicas configuradas de la siguiente forma:
 - Partición primaria: donde estará albergado el sistema operativo. Se le ha dado un almacenamiento de 20 GB. Para configurar esta partición debe escogerse el sistema de ficheros ext4 y “/” como punto de montaje.
 - Área de intercambio: donde se guardan temporalmente los archivos que no quepan en la memoria RAM. Se le ha dado unos 8 GB. Para configurar esta partición debe escogerse en “utilizar como” la opción “área de intercambio”.
 - Partición home: donde estarán almacenados los archivos. Se le ha dado el resto del área libre que quedaba, unos 77 GB. Para configurar esta partición debe escogerse también el sistema de ficheros ext4 y “/home” como punto de montaje.
- Pulsar el botón “Instalar ahora”, aparecerá una ventana para confirmar los cambios en el disco. Pulsar “continuar”.
- Escoger la zona horaria y especificar el nombre de usuario y contraseña para comenzar con el proceso de instalación.

Una vez terminada la instalación, cada vez que se inicie el ordenador aparecerá una ventana donde podrá elegirse el sistema operativo a utilizar.

3.2 Instalación y configuración de ROS

3.2.1 Instalación

Para la instalación de ROS *Melodic* [17], en primer lugar, se debe configurar el ordenador para que acepte software proveniente de packages.ros.org, ello se realiza introduciendo el siguiente comando en la terminal:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Luego debe configurarse las claves introduciendo:

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Una vez realizado esto, debe comprobarse que todos los paquetes estén actualizados mediante la introducción del siguiente comando:

```
sudo apt update
```

Si no fuera así, pueden instalarse estos paquetes introduciendo:

```
sudo apt upgrade
```

Más adelante, se presentan varias opciones para obtener ROS. De estas se adquiere la versión full, ya que no solo instala ROS, sino también el simulador Gazebo y la herramienta rqt que sirve de bastante ayuda para el estudio de los procesos en ROS, como las relaciones entre los distintos nodos, graficar datos provenientes de topics, etc. Para instalar esta versión se introduce el comando:

```
sudo apt install ros-melodic-desktop-full
```

3.2.2 Configuración del Entorno

En este momento ROS ya está instalado, pero antes de utilizarlo se debe seguir una serie de pasos para configurar el entorno de trabajo. En primer lugar, mencionar que para la ejecución de los comandos de ROS se debe introducir antes el siguiente comando por cada terminal donde se quiera ejecutar comandos ROS:

```
source /opt/ros/melodic/setup.bash
```

Ello es así porque ROS permite que en una máquina se instalen distintas versiones y mediante este comando se escoge la distribución a utilizar. Para evitar repetir este comando varias veces, puede añadirse al fichero “bashrc” introduciendo en la terminal el siguiente comando:

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
```

A continuación, se debe reiniciar la terminal para que se apliquen los cambios. El siguiente paso es crear un entorno de trabajo, para ello se debe introducir la siguiente secuencia de comandos:

- El primer comando crea el directorio `catkin_ws`, el cual contendrá el entorno de trabajo (el nombre se puede modificar). Dentro de este crea también la carpeta `src`, donde se incluirán todos los paquetes ROS desarrollados.
- El segundo comando cambia el directorio actual por el del nuevo entorno de trabajo.
- Finalmente se realiza una compilación, proceso que genera dos carpetas (`build` y `devel`) y un archivo `"CMakeLists.txt"` dentro de la carpeta `src`.

```
mkdir -p ~/catkin_ws/src  
cd ~/catkin_ws  
catkin_make
```

El siguiente comando añade al archivo `"bashrc"` una sentencia con tal de que ROS pueda encontrar los paquetes que se encuentren en el nuevo entorno de trabajo.

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

Por último, para crear el paquete ROS, se ha ejecutado el siguiente comando en la carpeta `src`:

```
catkin_create_pkg tfg_alvaro rospy
```

El primer campo se refiere al nombre del paquete (este nombre no debe contener caracteres especiales, mayúsculas o espacios) y el segundo campo es una lista opcional de dependencias a añadir a dicho paquete. En este caso se ha agregado únicamente la dependencia `rospy`, necesaria para el uso de nodos escritos en lenguaje Python. Si se quisiera cambiar más tarde esta lista de dependencias, se podrá realizar modificando el fichero `"CMakeLists.txt"` del paquete, generado en el momento de su creación.

3.3 Instalación del simulador SITL de ardupilot y el plugin de Gazebo

Para la instalación del simulador SITL de ardupilot [18] se deben seguir los siguientes pasos en una terminal. Primero clonamos el repositorio de github mediante la siguiente secuencia de comandos:

```
cd ~  
git clone https://github.com/ArduPilot/ardupilot.git  
cd ardupilot  
git submodule update --init --recursive
```

Instalamos los paquetes de Python: `future`, `pymavlink` y `MAVProxy`. Utilizando el administrador de paquetes `pip`.

```
sudo pip install future pymavlink MAVProxy
```

Finalmente añadimos las siguientes líneas al final del fichero `"bashrc"`:

```
export PATH=$PATH:$HOME/ardupilot/Tools/autotest
```



```
export PATH=/usr/lib/ccache:$PATH
```

En cuanto al plugin de Gazebo [19], existen dos versiones distintas: Khancyr y SwiftGust, sin embargo, las diferencias entre ellos radican en la documentación que aportan, siendo mayor en el segundo caso. De los dos se ha elegido Khancyr ya que es el original. Para la instalación del plugin se introduce la siguiente secuencia de comandos:

```
git clone https://github.com/khancyr/ardupilot_gazebo
cd ardupilot_gazebo
mkdir build
cd build
cmake ..
make -j4
sudo make install
```

3.4 Instalación del paquete MAVROS

Para la instalación del paquete MAVROS [20] se ejecuta el siguiente comando:

```
sudo apt-get install ros-melodic-mavros ros-melodic-mavros-extras
```

A continuación, se instala el conjunto de datos *GeographicLib* mediante el uso de los siguientes comandos:

```
wget
https://raw.githubusercontent.com/mavlink/mavros/master/mavros/scripts
/install_geographiclib_datasets.sh
chmod a+x install_geographiclib_datasets.sh
./install_geographiclib_datasets.sh
```

3.5 Instalación de QGroundControl

Antes de la instalación de QGroundControl, se deben introducir los siguientes comandos:

```
sudo usermod -a -G dialout $USER
sudo apt-get remove modemmanager -y
sudo apt install gstreamer1.0-plugins-bad gstreamer1.0-libav
gstreamer1.0-gl -y
```

Para la instalación de QGroundControl se descarga la app Image que se puede encontrar en su página web [21], y en la terminal introducir el siguiente comando para convertir el archivo en ejecutable:

```
chmod +x ./QGroundControl.AppImage
```

Luego para ejecutar la aplicación puede utilizarse el siguiente comando o simplemente doble clic sobre el archivo descargado.

```
./QGroundControl.AppImage
```

4. Implementación

4.1 Inicio del entorno de simulación

Una vez se ha instalado todo correctamente, se procede a inicializar el entorno de simulación. En primer lugar, se ejecuta Gazebo con el escenario “iris_arducopter_runway.world” (el cual trae el plugin por defecto) mediante el uso del siguiente comando:

```
gazebo --verbose worlds/iris_arducopter_runway.world
```

Un escenario o mundo es un archivo que lleva por extensión “world”, en el cual se define en formato XML los diferentes parámetros físicos y gráficos que debe tomar el simulador, así como los modelos que se emplean. El entorno de simulación utilizado en este proyecto está formado por una pista de aterrizaje (que simula el campo del club de aeromodelismo de Canberra, Australia) y el modelo de un dron Iris de la marca 3D Robotics, tal y como se observa en la *Figura 16*.

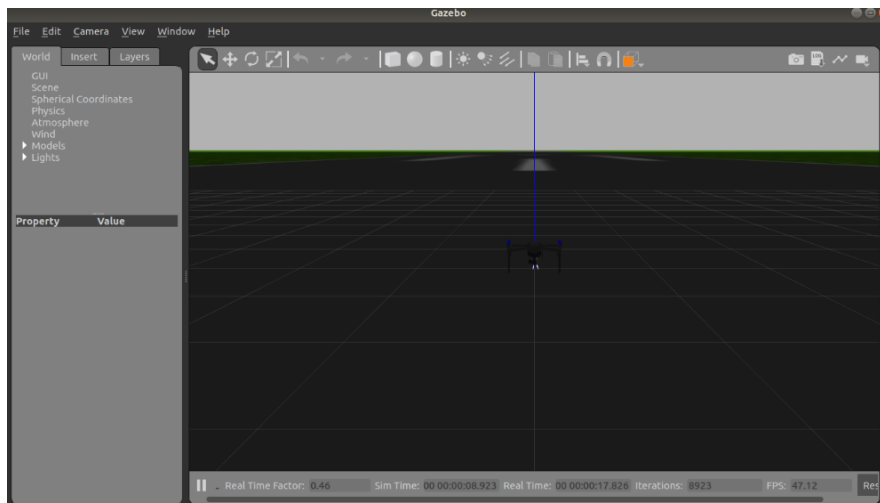


Figura 16: Gazebo con el escenario iris_arducopter_runway.world.

En ocasiones al intentar cargar el escenario, puede acabar resultando en un error como el que se muestra en la *Figura 17*, especialmente cuando ya se ha iniciado el simulador con anterioridad. Este error se debe principalmente a que algún proceso de Gazebo no ha conseguido cerrarse del todo y sigue ejecutándose. Para poder solucionar este problema se debe poner fin a todos estos procesos mediante el comando:

```
killall gzserver
```

```
alvaro@alvaro-Aspire-F5-573G: ~  
Archivo Editar Ver Buscar Terminal Ayuda  
alvaro@alvaro-Aspire-F5-573G:~$ gazebo --verbose worlds/iris_arducopter_runway.w  
orld  
Gazebo multi-robot simulator, version 9.16.0  
Copyright (C) 2012 Open Source Robotics Foundation.  
Released under the Apache 2 License.  
http://gazebo.org  
[Err] [Master.cc:96] EXCEPTION: Unable to start server[bind: Address already in  
use]. There is probably another Gazebo process running.  
[Err] [Master.cc:96] EXCEPTION: Unable to start server[bind: Address already in  
use]. There is probably another Gazebo process running.  
alvaro@alvaro-Aspire-F5-573G:~$
```

Figura 17: Error al iniciar Gazebo.

Para iniciar SITL junto con MAVProxy se introduce en la terminal la siguiente secuencia de comandos:

```
cd ~/ardupilot/ArduCopter  
../Tools/autotest/sim_vehicle.py -f gazebo-iris --console --map
```

Una vez iniciado el simulador, la terminal se quedará a la espera de comandos para poder controlar el dron. También aparece un mapa con la ubicación del UAV y una consola que muestra información de la aeronave como: su nivel de batería, modo de funcionamiento, estado de los sensores, etc. Tal y como se observa en la Figura 18.

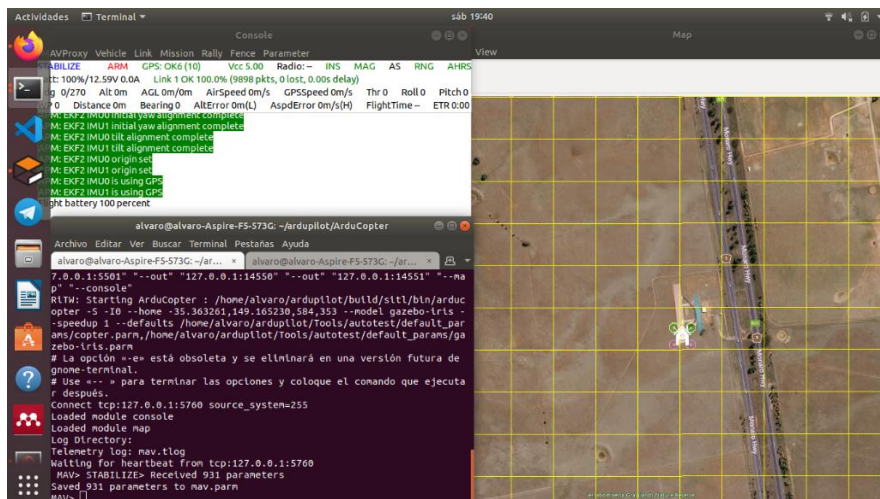


Figura 18: Ejecución del simulador SITL junto a MAVProxy.

4.1.1 Prueba del simulador con MAVProxy

4.1.1.1 Despegue

A continuación, se realiza una prueba del funcionamiento del simulador haciendo uso del software GCS MAVProxy. En primer lugar, se comienza con el despegue de la aeronave introduciendo la siguiente secuencia de comandos en la terminal:

- Cambio de modo de vuelo del UAV a guiado, permitiendo el control del dron desde MAVProxy o cualquier otro GCS.

mode guided

- Preparación de los motores para el despegue. En Gazebo se observa cómo después de introducir dicho comando, los motores empiezan a girar mientras en la esquina superior izquierda de la consola se advierte como la palabra ARM ha cambiado de color rojo a verde.

arm throttle

- Se introduce el comando “takeoff” junto con la altura en metros a la que se quiere elevar el dron (Figura 19). Este comando debe introducirse rápidamente después del armado, ya que los motores se paran si no se realiza una acción durante un tiempo.

takeoff 10

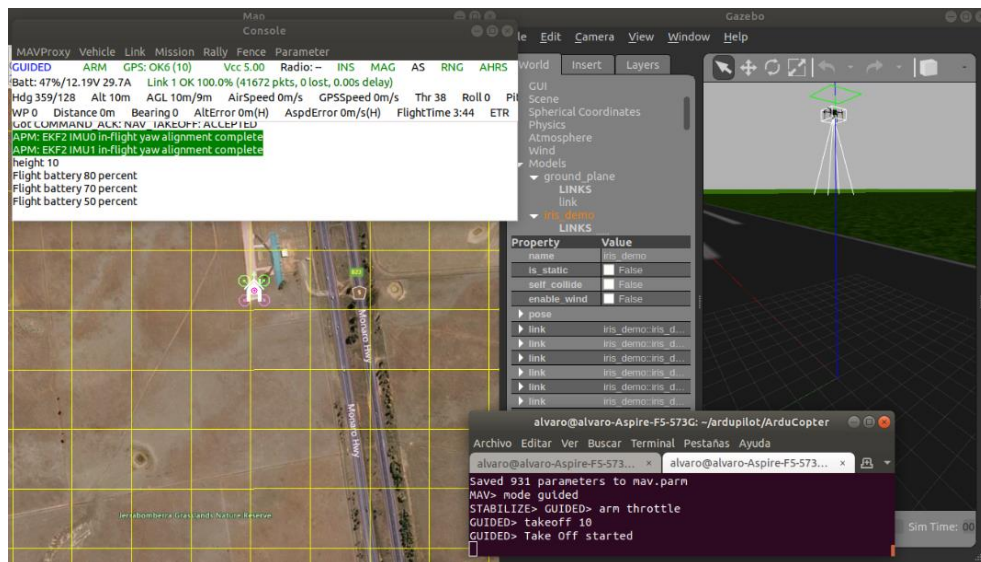


Figura 19: Secuencia de despegue del UAV.

4.1.1.2 Movimiento simple a un punto

Una vez esté el dron en el aire, ya puede ser controlado. Para conseguir que el UAV se dirija a un punto concreto, se deberá seguir los siguientes pasos:

- Desde el mapa, pulsar clic derecho en el punto destino y escoger la opción “Fly to”.
- Aparecerá una ventana donde se deberá especificar la altura a la que se encuentra dicho punto.

En la Figura 20, se puede observar como el dron está intentando llegar a un punto destino introducido en el mapa, que se encuentra a una altura de 40 metros.

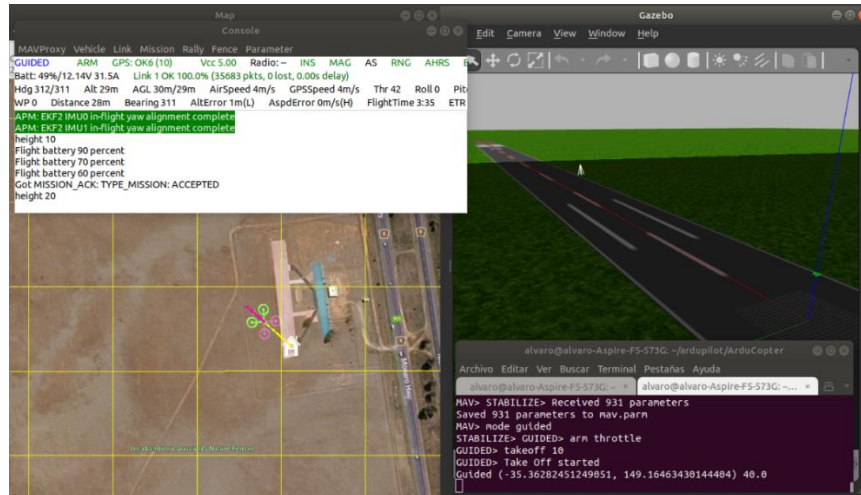


Figura 20: UAV dirigiéndose a un punto introducido.

4.1.1.3 Cambio de base

Otra de las opciones que brinda MAVProxy, es la de cambiar la base o punto de despegue, representado en el mapa con el icono de una casa. Para ello, de la misma manera que en el caso anterior, se debe pulsar clic derecho sobre un punto del mapa y seleccionar esta vez la opción “Set Home” (Figura 21).

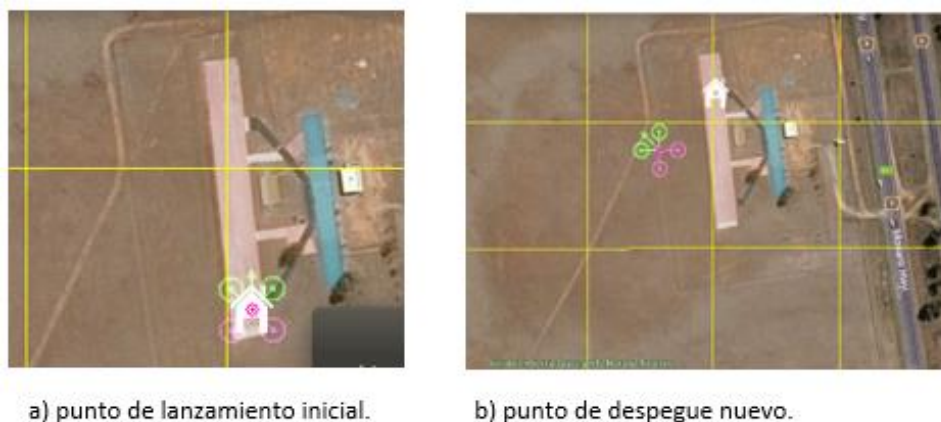


Figura 21: Cambio del punto de despegue.

4.1.1.4 Planificación de misiones

Con MAVProxy se puede también planificar trayectorias mediante la definición de puntos de referencia o también conocidos como waypoints. Para ello se deberá seguir los siguientes pasos:

- Desde el mapa, pulsar clic derecho en el punto destino y escoger la opción “Mission” y seguidamente “Draw”.
- Pulsar clic izquierdo sobre el mapa para añadir puntos a la trayectoria. En la Figura 22 puede observarse como se realiza una trayectoria de prueba.

- Pulsar clic derecho sobre cualquier parte del mapa para terminar de dibujar la trayectoria.
- Aparecerá una ventana que pedirá introducir la altura en metros a la que se quiere ejecutar la misión.

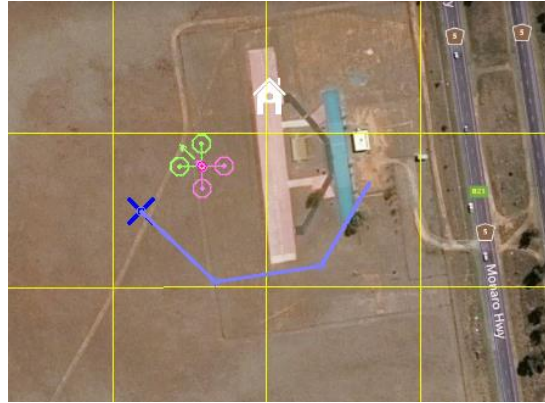


Figura 22: Realización de una trayectoria en MAVProxy.

Para comenzar con la misión basta con cambiar al modo de vuelo auto mediante el uso del siguiente comando:

```
mode auto
```

El dron comenzará a dirigirse al primer punto de la trayectoria y una vez esté lo bastante cerca pasará automáticamente al siguiente, repitiendo el proceso hasta terminar la trayectoria (Figura 23).

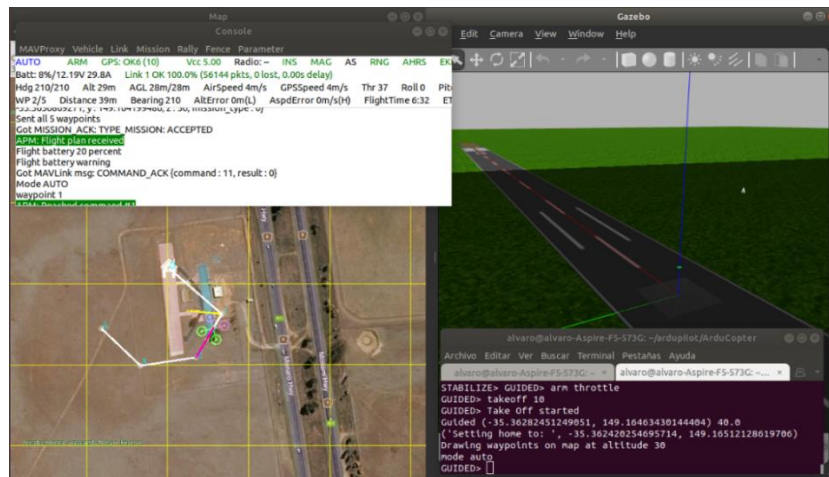


Figura 23: UAV siguiendo una trayectoria definida.

4.1.1.5 Aterrizaje

Finalmente se procede al aterrizaje del dron. Para ello existen dos modos distintos: RTL y Land. Para activar uno de estos modos se debe introducir los siguientes comandos:

```
mode rtl
mode land
```

- El modo Land consiste de un aterrizaje in situ. Al activar dicho modo, el dron desde la posición en la que se encuentra, comienza a bajar hasta el suelo donde finalmente desarma los motores. De tal forma que el punto de aterrizaje puede ser distinto al punto base, tal y como se observa en la *Figura 24*.

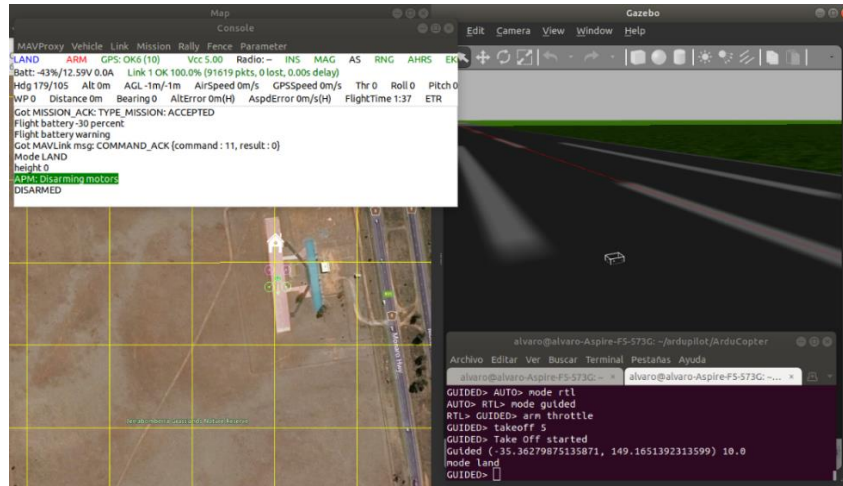
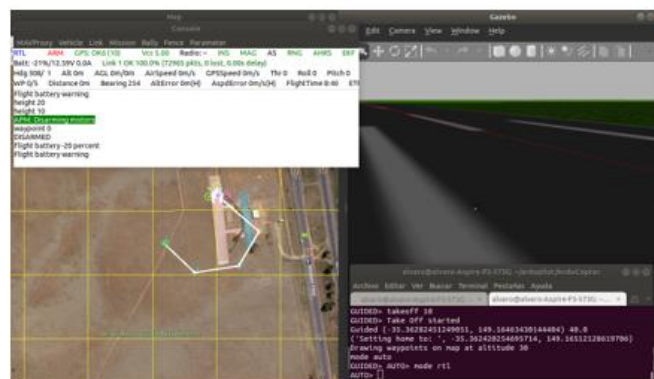


Figura 24: Aterrizaje en modo Land.

- El modo RTL o de vuelta al punto de lanzamiento (*Return to Launch*), es un modo parecido al anterior con la salvedad de que antes de comenzar con el aterrizaje primero el dron se dirigirá hasta el punto de despegue. En la *Figura 25* se muestra este funcionamiento.



a) Dron dirigiéndose al punto de despegue.



b) UAV finalmente aterrizando.

Figura 25: Aterrizaje en modo RTL.

4.2 Estudio de QGroundControl

Al iniciar QGroundControl se obtiene una interfaz como la de la *Figura 26*. Se puede observar al UAV representado con una flecha roja sobre el mapa. En la parte inferior se muestran varios datos de importancia como son la altura, velocidad, tiempo de vuelo y distancia recorrida. Esta ventana de detalles puede modificarse pulsando doble clic sobre ella, pudiendo así añadir o eliminar columnas y filas al mismo tiempo que incluir otro tipo de datos como por ejemplo velocidad del viento. Por otro lado, en la parte superior derecha se presenta el autopiloto que gobierna el dron, un indicador de actitud y una brújula. Por último, en el lado izquierdo se encuentra la escala del mapa y otros datos como el nivel de batería y el modo de vuelo, además de una pestaña con algunas opciones para controlar el dron.

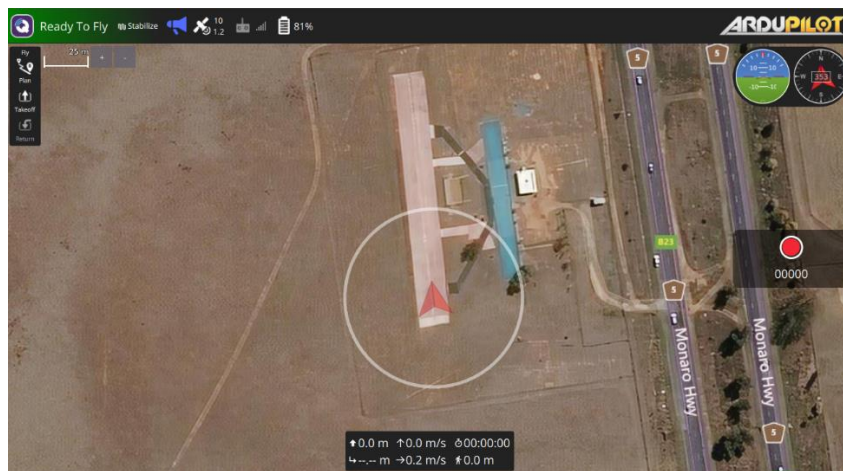


Figura 26: QGroundControl.

4.2.1 Despegue

Para despegar el dron se deberá:

- Pulsar la opción "Takeoff".
- Aparecerá un indicador de altitud en el lateral derecho de la pantalla junto con un mensaje de confirmación en la parte inferior (*Figura 27*).
- Una vez deslizada la flecha hacia la derecha, el dron comenzará a elevarse hasta la altura indicada.

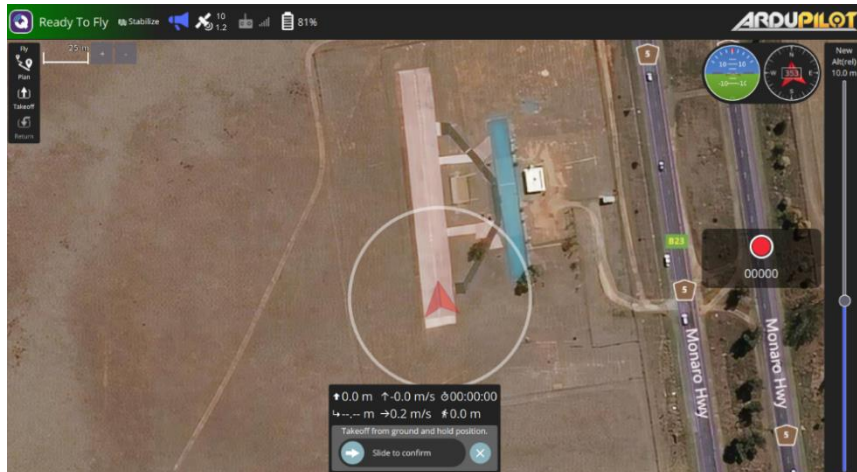


Figura 27: Confirmación de despegue del UAV.

4.2.2 Movimiento simple a un punto

Ya con el dron en el aire, para conseguir que se dirija hacia un punto en concreto del mapa, deberá seguirse los siguientes pasos:

- Pulsar clic izquierdo sobre la localización deseada y escoger la opción “Go to Localization” (la única que aparece).
- Como en el caso anterior, se mostrará un mensaje de confirmación.
- Una vez se apruebe dicha orden, el dron comenzará a dirigirse hacia el punto señalado sin cambiar su altura (Figura 28).

También destaca la aparición (en la pestaña de comandos) de las opciones “Land” y “Return” para aterrizar el dron tanto en modo Land como RTL respectivamente, además de la opción “Pause” que permite parar el dron mientras esté en movimiento.



Figura 28: UAV dirigiéndose a un punto especificado en QGroundControl.

4.2.3 Misiones en QGroundControl

4.2.3.1 Planificación

En QGroundControl también pueden elaborarse planes de vuelo de manera sencilla. Para ello se debe:

- Seleccionar la opción “Plan” en la pestaña de comandos.
- Aparecerá el menú de la *Figura 29*. En este menú se muestran varios modelos de misiones realizados con distintos patrones que ofrece QGroundControl y que más tarde serán explicados.

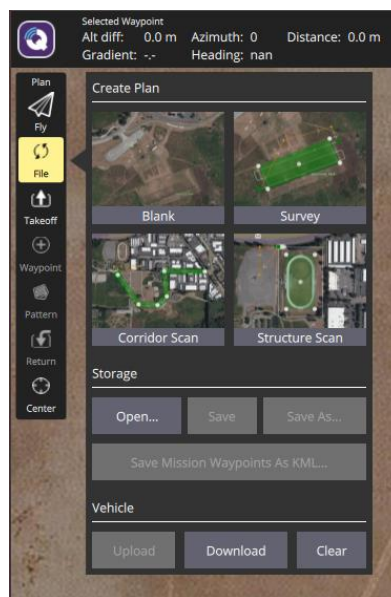


Figura 29: Menú de creación de misiones.

- De estas opciones se escoge “Blank” para definir la trayectoria desde cero.
- Una vez seleccionado el modelo, se abrirá el editor de misiones conformado por un conjunto de opciones en la parte lateral izquierda con varios comandos de vuelo.
- En QGroundControl toda misión debe tener como mínimo al principio un comando de despegue, por ello se debe seleccionar inicialmente la opción “Takeoff”.
- Para añadir puntos a la trayectoria, hay que escoger la opción “WayPoint” y pulsar clic izquierdo sobre el mapa.
- Mediante la opción “Return” se añade un comando de aterrizaje RTL. En la *Figura 30* se muestra un ejemplo de trayectoria realizada en QGroundControl.

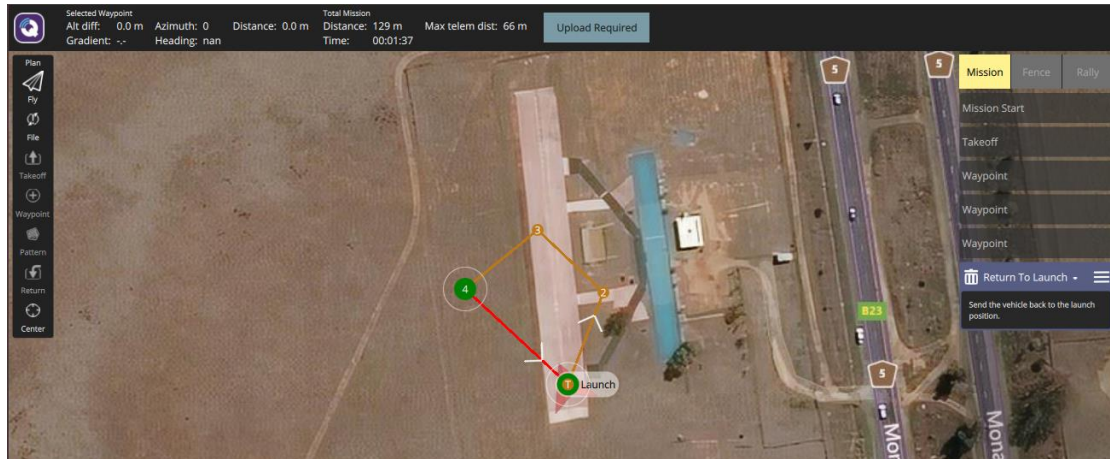
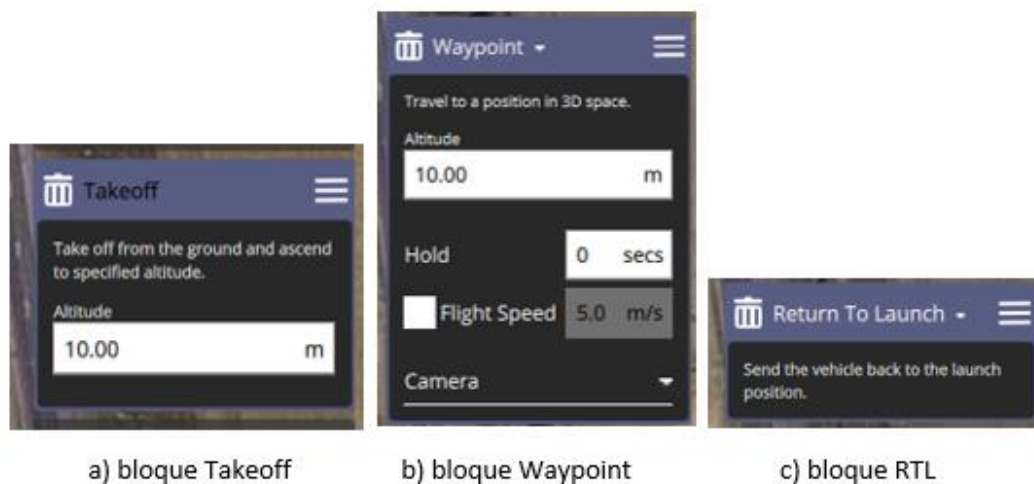


Figura 30: trayectoria definida en QGroundControl.

- En la parte lateral derecha, aparecerán una serie de paneles. Estos se añaden cada vez que se especifique un comando o agregue un punto a la trayectoria:
 - Pulsando el bloque de despegue (Figura 31.a) puede modificarse la altura a la que asciende el dron.
 - Seleccionando cualquiera de los paneles correspondientes a cada punto (Figura 31.b) puede especificarse tanto la altura a la que se encuentra dicho punto como el tiempo en segundos que permanezca allí la aeronave, la velocidad de salida y controlar la cámara: tomar fotos, iniciar/parar grabación, etc. También es posible eliminar dicho punto seleccionando el icono con forma de papelera.
 - El panel RTL (Figura 31.c) no especifica ningún parámetro modificable, sin embargo pulsando el nombre del bloque, es posible cambiar el modo de aterrizaje a Land.



a) bloque Takeoff

b) bloque Waypoint

c) bloque RTL

Figura 31: Bloques de configuración de parámetros.

- Ya definida la trayectoria, pulsar el botón “upload” en la parte superior de la pantalla para cargar esta en el dron.
- Si la misión se ha subido con éxito se mostrará el mensaje “Done”.

4.2.3.2 Ejecución de trayectorias

Una vez elaborada la trayectoria, para iniciar la misión hay que:

- Volver a la pantalla de vuelo seleccionando la opción “Fly”, que tiene el símbolo de un avión de papel.
- Aparecerá un mensaje de confirmación como los anteriormente mencionados.
- Una vez aceptada la orden, el dron comenzará a seguir la trayectoria establecida tal y como se observa en la *Figura 32*.
- Si en medio del recorrido se parase manualmente la aeronave o se produjese algún problema, se puede reanudar la misión seleccionando la opción “Action” de la parte lateral izquierda.



Figura 32: UAV siguiendo la trayectoria definida.

4.2.3.3 Tipos de patrones

QGroundControl ofrece además una manera sencilla y automática de crear misiones más complejas solo cambiando un par de parámetros. Volviendo al editor de misiones, si se selecciona la opción “Patter” (*Figura 33*), aparecerán tres tipos de patrones distintos: survey, corredor scan y structure scan.

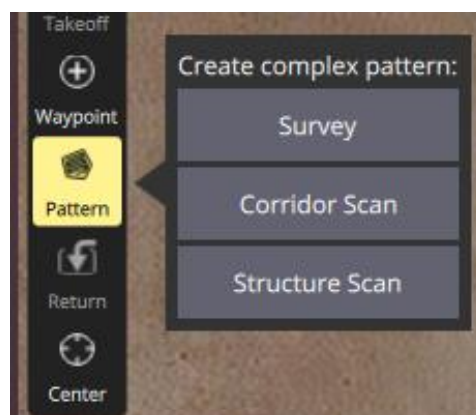


Figura 33: Tipos de patrones.

- Survey permite realizar un reconocimiento de un area especificada, definiendo una trayectoria tipo zigzag alrededor de esta, tal y como se observa en la *Figura 34*. Esta

superficie puede ser tanto circular como poligonal siendo modificable su extensión y forma, también mediante su panel correspondiente se puede cambiar la altura a la que se realiza el análisis, así como el espacio que recorre entre cada paso, etc.



Figura 34: Ejemplo de misión con un patrón del tipo survey.

- Corridor scan permite realizar un escaneo a lo largo de un determinado recorrido, realizando uno o varios viajes de ida y vuelta. Dicho trayecto estará formado por un único tramo recto (opción "Basic") o una sucesión de estos (opción "Trace") tal y como se observa en la Figura 35. En el mapa puede controlarse tanto la cantidad de tramos como el largo de cada uno y en su respectivo panel modificar parámetros como la altura del recorrido o el ancho del mismo.



Figura 35: Ejemplo de misión con un patrón del tipo corridor scan.

- Structure scan (Figura 36) permite realizar el escaneo de una determinada estructura, ya sea un edificio o monumento. Para ello se debe definir un área de escaneo y QGroundControl calculará automáticamente una trayectoria alrededor de esta. Dicha superficie, como en el caso del patrón survey, puede ser tanto circular como poligonal pudiendo variar su extensión y forma en el mapa. Mediante su panel correspondiente puede modificarse parámetros como la altura de la estructura a escanear o la distancia a la que se realiza el escaneo.

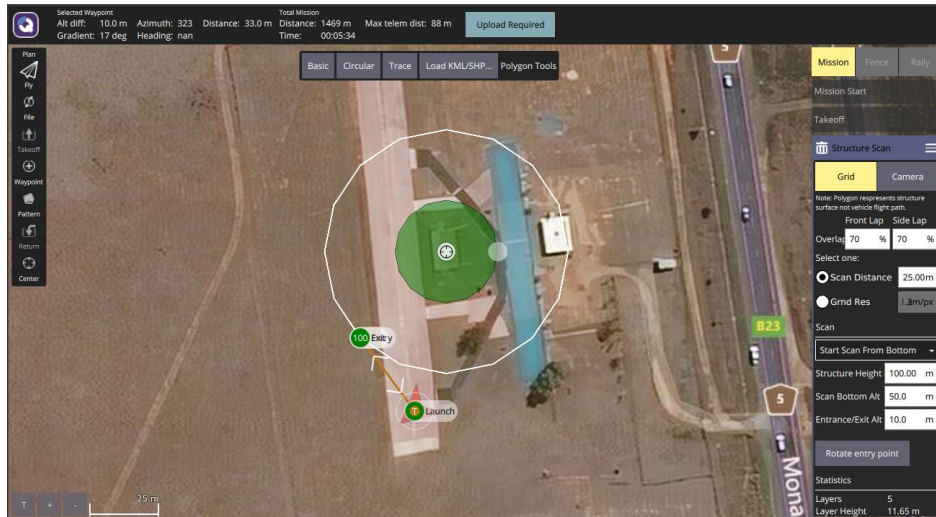


Figura 36: Ejemplo de misión con un patrón del tipo structure scan.

4.3 MAVROS

Una vez ejecutado el simulador y comprobado que todo funciona correctamente. El siguiente paso es iniciar el nodo MAVROS. Para ello el paquete cuenta con el archivo “*amp.launch*”, ejecutable mediante el comando *roslaunch*. Pero antes, para que el nodo pueda comunicarse con el simulador, debe aplicarse una modificación [22] a dicho archivo. Una buena práctica en estos casos es clonar el fichero en otro paquete y realizar las modificaciones sobre dicho clon, de esta forma el paquete de MAVROS queda siempre intacto. Para clonar el archivo se ha introducido la siguiente secuencia de comandos, donde primero se cambia el directorio actual por el del paquete *tfg_alvaro*, allí crea el directorio *launch* y mediante el comando *roscp*, clona el fichero en esa carpeta.

```
roscd tfg_alvaro
mkdir launch
cd launch
roscp mavros apm.launch apm.launch
```

Ahora con el simulador iniciado, en su terminal correspondiente puede encontrarse una línea, resaltada en la Figura 37, donde se definen los puertos de acceso al simulador mediante el protocolo UDP, creados por MAVProxy.

```
7.0.0.1:5501 "--out" "127.0.0.1:14550" "--out" "127.0.0.1:14551" "--ma
opter -S -I0 --home -35.363261,149.165230,584,353 --model gazebo-iris -
ams/copter.parm,/home/alvaro/ardupilot/Tools/autotest/default_params/ga
```

Figura 37: Puertos de conexión con el simulador.

Teniendo esto en cuenta, se ha modificado la primera línea de “*apm.launch*” por la siguiente, quedando el archivo como se muestra en la Figura 38.

```
<arg name="fcu_url" default="udp://127.0.0.1:14551@14555" />
```

```

Abrir  vim: set ft=xml noet : -->
*apm.launch
~/catkin_ws/src/tfg_alvaro/launch

<!-- vim: set ft=xml noet : -->
<!-- example launch script for ArduPilot based FCU's -->

<arg name="fcu_url" default="udp://127.0.0.1:14551@14555" />
<arg name="gcs_url" default="" />
<arg name="tgt_system" default="1" />
<arg name="tgt_component" default="1" />
<arg name="log_output" default="screen" />
<arg name="fcu_protocol" default="v2.0" />
<arg name="respawn_mavros" default="false" />

<include file="$(find mavros)/launch/node.launch">
  <arg name="pluginlists_yaml" value="$(find mavros)/launch/apm_pluginlists.yaml" />
  <arg name="config_yaml" value="$(find mavros)/launch/apm_config.yaml" />

  <arg name="fcu_url" value="$(arg fcu_url)" />
  <arg name="gcs_url" value="$(arg gcs_url)" />
  <arg name="tgt_system" value="$(arg tgt_system)" />
  <arg name="tgt_component" value="$(arg tgt_component)" />
  <arg name="log_output" value="$(arg log_output)" />
  <arg name="fcu_protocol" value="$(arg fcu_protocol)" />
  <arg name="respawn_mavros" value="$(arg respawn_mavros)" />
</include>
</launch>

```

Figura 38: Archivo “apm.launch” modificado.

Una vez modificado “apm.launch”, para lanzar MAVROS se introduce el siguiente comando desde terminal:

```
roslaunch tfg_alvaro apm.launch
```

En la Figura 39 se comprueba que el nodo este activo mediante el comando *roslaunch tfg_alvaro apm.launch*, también se obtiene más información sobre dicho nodo a través de *roslaunch tfg_alvaro apm.launch*. Llama la atención la cantidad enorme de topics y servicios ofrecidos por dicho nodo, de los cuales, para el desarrollo de un software de control solo son realmente importantes unos cuantos. Este hecho acentúa más la necesidad de crear un nodo interfaz que además de automatizar algunas funciones, también sea capaz de filtrar los topics más interesantes y así hacer más sencillo la creación de un posterior nodo de control del UAV.

```

alvaro@alvaro-Aspire-F5-573G:~$ roslaunch tfg_alvaro apm.launch
/mavros
/rosout
alvaro@alvaro-Aspire-F5-573G:~$ roslaunch tfg_alvaro apm.launch
-----
Node [/mavros]
Publications:
* /diagnostics [diagnostic_msgs/DiagnosticArray]
* /mavlink/from [mavros_msgs/Mavlink]
* /mavlink/gcs_ip [std_msgs/String]
* /mavros/adsb/vehicle [mavros_msgs/ADSBVehicle]
* /mavros/battery [sensor_msgs/BatteryState]
* /mavros/cam_imu_sync/cam_imu_stamp [mavros_msgs/CamIMUStamp]
* /mavros/distance_sensor/rangefinder_pub [sensor_msgs/Range]
* /mavros/esc_info [mavros_msgs/ESCInfo]
* /mavros/esc_status [mavros_msgs/ESCStatus]
* /mavros/estimator_status [mavros_msgs/EstimatorStatus]
* /mavros/extended_state [mavros_msgs/ExtendedState]
* /mavros/global_position/compass_hdg [std_msgs/Float64]
* /mavros/global_position/global [sensor_msgs/NavSatFix]
* /mavros/global_position/gp_lp_offset [geometry_msgs/PoseStamped]
* /mavros/global_position/gp_origin [geographic_msgs/GeoPointStamped]
* /mavros/global_position/local [nav_msgs/Odometry]
* /mavros/global_position/raw/fix [sensor_msgs/NavSatFix]

```

Figura 39: Lista de nodos activos e información sobre el nodo mavros.

De esa cantidad enorme de topics, se han considerado interesantes para su uso en este proyecto los siguientes:

- /mavros/battery: aquí mavros publica mensajes del tipo “BatteryState” de “sensor_msgs” con el estado de la batería de la aeronave.

- /mavros/global_position/global: posición del dron global en coordenadas geográficas (latitud, longitud y altitud) obtenida del GPS, los mensajes son del tipo “NavSatFix” de la dependencia “sensor_msgs”.
- /mavros/local_position/pose: en este topic mavros publica la pose actual del dron (posición y orientación) con el formato “PoseStamped” de “geometry_msgs” con respecto al punto de despegue.
- /mavros/local_position/velocity_body: aquí mavros publica la velocidad local de la aeronave con respecto a su propio sistema de referencia (Figura 40). Este topic utiliza mensajes del tipo “TwistStamped” de la dependencia “geometry_msgs” y será utilizado más adelante en la parte del modelado para medir la respuesta del dron ante estímulos de velocidad.

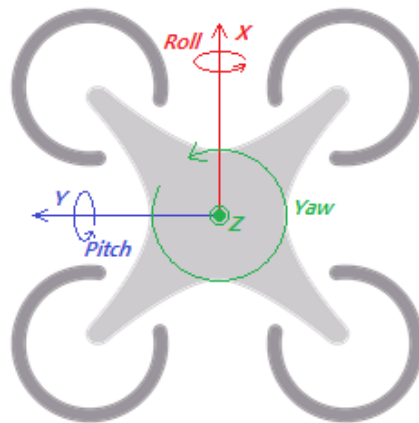


Figura 40: Eje de referencia local de la aeronave.

- /mavros/state: en este topic mavros publica información sobre el dron como puede ser el modo de vuelo o el estado de los motores, los mensajes son del tipo “State” de la dependencia “mavros_msgs”.
- /mavros/home_position/set: mediante este topic se puede cambiar el punto de despegue, enviando mensajes del tipo “mavros_msgs/HomePosition”. Será utilizado más adelante en el nodo interfaz para mantener este punto base siempre en la misma posición desde la que se inició el dron por primera vez.
- /mavros/setpoint_position/local: en este topic podemos enviar mensajes del tipo “geometry_msgs/PoseStamped” para especificar consignas de posición y orientación con respecto a un sistema de referencia local definido al iniciar el dron.
- /mavros/setpoint_velocity/cmd_vel: mediante este topic podemos enviar consignas de velocidad al dron utilizando mensajes del tipo “geometry_msgs/TwistStamped”, por ello será utilizado para la parte del modelado.

En cuanto a los servicios se han considerado relevantes los mostrados a continuación:

- /mavros/set_mode: este servicio permite cambiar el modo de vuelo del UAV, por tanto, será utilizado tanto en el despegue como en el aterrizaje del dron de la misma forma que se enseñó anteriormente con MAVProxy (apartado 4.1.1). Tal y como se observa en la Figura 41, este servicio es del tipo “mavros_msgs/SetMode” el cual tiene dos argumentos de entrada, base_mode y custom_mode donde debe especificarse el nuevo modo de vuelo. Como respuesta devuelve una variable booleana que simboliza el éxito de la orden.


```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice info /mavros/set_mode
Node: /mavros
URI: rosrpc://alvaro-Aspire-F5-573G:43703
Type: mavros_msgs/SetMode
Args: base_mode custom_mode
alvaro@alvaro-Aspire-F5-573G:~$ rossrv show mavros_msgs/SetMode
uint8 MAV_MODE_PREFLIGHT=0
uint8 MAV_MODE_STABILIZE_DISARMED=80
uint8 MAV_MODE_STABILIZE_ARMED=208
uint8 MAV_MODE_MANUAL_DISARMED=64
uint8 MAV_MODE_MANUAL_ARMED=192
uint8 MAV_MODE_GUIDED_DISARMED=88
uint8 MAV_MODE_GUIDED_ARMED=216
uint8 MAV_MODE_AUTO_DISARMED=92
uint8 MAV_MODE_AUTO_ARMED=220
uint8 MAV_MODE_TEST_DISARMED=66
uint8 MAV_MODE_TEST_ARMED=194
uint8 base_mode
string custom_mode
---
bool mode_sent
alvaro@alvaro-Aspire-F5-573G:~$
```

Figura 41: Información del servicio /mavros/set_mode.

- /mavros/cmd/arming: este servicio es utilizado para el armado de los motores. Como se observa en la Figura 42, es del tipo “mavros_msgs/CommandBool”. Como argumento de entrada dispone de la variable booleana “value”, que representa el estado deseado de los motores (True para armado y False para desarmado).

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice info /mavros/cmd/arming
Node: /mavros
URI: rosrpc://alvaro-Aspire-F5-573G:43703
Type: mavros_msgs/CommandBool
Args: value
alvaro@alvaro-Aspire-F5-573G:~$ rossrv show mavros_msgs/CommandBool
bool value
---
bool success
uint8 result
```

Figura 42: Información del servicio /mavros/cmd/arming.

- /mavros/cmd/takeoff: este servicio es utilizado para despegar el dron a la altura especificada, una vez estén armados los motores. En la Figura 43 se observa que este servicio es del tipo “mavros_msgs/CommandTOL” y que solicita como entrada varios datos, de los cuales solo es realmente importante “altitude”, la altitud a la que se quiere despegar el dron.

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice info /mavros/cmd/takeoff
Node: /mavros
URI: rosrpc://alvaro-Aspire-F5-573G:43703
Type: mavros_msgs/CommandTOL
Args: min_pitch yaw latitude longitude altitude
alvaro@alvaro-Aspire-F5-573G:~$ rossrv show mavros_msgs/CommandTOL
float32 min_pitch
float32 yaw
float32 latitude
float32 longitude
float32 altitude
---
bool success
uint8 result
```

Figura 43: Información del servicio /mavros/cmd/takeoff.

- /mavros/setpoint_velocity/mav_frame: mediante este servicio puede modificarse el sistema de referencia al cual se refieren las consignas de velocidad enviadas por el topic /mavros/setpoint_velocity/cmd_vel. En la Figura 44 se muestra información sobre este servicio, donde se comprueba que es del tipo “mavros_msgs/SetMavFrame”,

demandando como parámetro de entrada el sistema de referencia nuevo. Este servicio será utilizado más adelante para cambiar el sistema de referencia por defecto, FRAME_LOCAL_NED, por el local a la aeronave, FRAME_BODY_NED (Figura 40), debido a que se ha considerado que mediante este último el control es más intuitivo.

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice info /mavros/setpoint_velocity/mav_frame
Node: /mavros
URI: rosrpc://alvaro-Aspire-F5-573G:43703
Type: mavros_msgs/SetMavFrame
Args: mav_frame
alvaro@alvaro-Aspire-F5-573G:~$ rossrv show mavros_msgs/SetMavFrame
uint8 FRAME_GLOBAL=0
uint8 FRAME_LOCAL_NED=1
uint8 FRAME_MISSION=2
uint8 FRAME_GLOBAL_RELATIVE_ALT=3
uint8 FRAME_LOCAL_ENU=4
uint8 FRAME_GLOBAL_RELATIVE_ALT_INT=5
uint8 FRAME_LOCAL_OFFSET_NED=7
uint8 FRAME_BODY_NED=8
uint8 FRAME_BODY_OFFSET_NED=9
uint8 FRAME_GLOBAL_TERRAIN_ALT=10
uint8 FRAME_GLOBAL_TERRAIN_ALT_INT=11
uint8 FRAME_BODY_FRD=12
uint8 FRAME_BODY_FLU=13
uint8 FRAME_MOCAP_NED=14
uint8 FRAME_MOCAP_ENU=15
uint8 FRAME_VISION_NED=16
uint8 FRAME_VISION_ENU=17
uint8 FRAME_ESTIM_NED=18
uint8 FRAME_ESTIM_ENU=19
uint8 mav_frame
---
bool success
```

Figura 44: Información del servicio /mavros/setpoint_velocity/mav_frame.

- /mavros/mission/clear: Este servicio es utilizado para borrar la misión actual que tiene cargada el dron. Es del tipo “mavros_msgs/WaypointClear” (Figura 45) el cual no especifica ningún argumento de entrada, pero si devuelve una variable booleana como respuesta, correspondiente al éxito de la orden.

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice info /mavros/mission/clear
Node: /mavros
URI: rosrpc://alvaro-Aspire-F5-573G:33161
Type: mavros_msgs/WaypointClear
Args:
alvaro@alvaro-Aspire-F5-573G:~$ rossrv show mavros_msgs/WaypointClear
---
bool success
```

Figura 45: Información del servicio /mavros/ misión/clear.

- /mavros/mission/push: este servicio permite enviar una misión al UAV. Como se observa en la Figura 46, es del tipo “mavros_msgs/WaypointPush” el cual requiere como argumento de entrada una lista de comandos del tipo “mavros_msgs/Waypoints”. Finalmente, como respuesta devuelve tanto el éxito de la orden, como el número de waypoints enviados.

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice info /mavros/mission/push
Node: /mavros
URI: rosrpc://alvaro-Aspire-F5-573G:33161
Type: mavros_msgs/WaypointPush
Args: start_index waypoints
alvaro@alvaro-Aspire-F5-573G:~$ rossrv show mavros_msgs/WaypointPush
uint16 start_index
mavros_msgs/Waypoint[] waypoints
  uint8 FRAME_GLOBAL=0
  uint8 FRAME_LOCAL_NED=1
  uint8 FRAME_MISSION=2
  uint8 FRAME_GLOBAL_REL_ALT=3
  uint8 FRAME_LOCAL_ENU=4
  uint8 frame
  uint16 command
  bool is_current
  bool autocontinue
  float32 param1
  float32 param2
  float32 param3
  float32 param4
  float64 x_lat
  float64 y_long
  float64 z_alt
  ...
bool success
uint32 wp_transferred
```

Figura 46: Información del servicio /mavros/misión/push.

4.4 Elaboración de la interfaz

Una vez teniendo claro el funcionamiento de MAVROS y habiendo seleccionado los topics y servicios más interesantes, se comenzó con el desarrollo de la interfaz. Para ello, se ha dividido su funcionamiento en dos nodos: `copter_node` y `misión_node`. Si bien se podría elaborar lo mismo con un único programa, se ha considerado preferible esta opción ya que facilita su comprensión. Estos nodos se han elaborado utilizando el estilo de programación orientado a objetos (mediante clases y objetos). El beneficio que trae consigo es un código más limpio, entendible y elegante. Además, han sido utilizadas sentencias `try-except` para el manejo de errores, incrementando así la robustez de la interfaz.

4.4.1 Copter_node

La principal tarea de este nodo es el filtrado de mensajes de MAVROS, así como la automatización y mejora de algunas funciones para conseguir un entorno de control del dron más limpio e intuitivo. Para su correcto funcionamiento, se deberá ejecutar luego de MAVROS y siempre antes de mover el dron por primera vez. En la *Figura 47*, puede observarse tanto la lista de topics como de servicios definidos para este nodo. Cada uno de los nombres de los topics y servicios comienza con la expresión `"/copter_node/"`. El motivo por el que se emplea esta distinción es el de obtener una estructura de mensajes más organizada.

```
alvaro@alvaro-Aspire-F5-573G:~$ rostopic list /copter_node/  
/copter_node/Battery  
/copter_node/Global_position  
/copter_node/Local_position  
/copter_node/SetPosition  
/copter_node/SetVelocity  
/copter_node/State  
/copter_node/Velocity  
alvaro@alvaro-Aspire-F5-573G:~$ rosservice list /copter_node/  
/copter_node/Guided  
/copter_node/Land  
/copter_node/RTL  
/copter_node/Stop  
/copter_node/Takeoff  
/copter_node/get_loggers  
/copter_node/set_logger_level  
alvaro@alvaro-Aspire-F5-573G:~$
```

Figura 47: Topics y servicios del nodo *copter_node*.

A continuación, se detalla el funcionamiento de cada uno de estos topics:

- Publicación:
 - /copter_node/Local_position: en este topic, copter_node publica la pose local del UAV con respecto al punto de despegue, recibida del topic /mavros/local_position/pose.
 - /copter_node/Global_position: aquí se publica la posición global del dron en coordenadas geográficas, recibida de /mavros/global_position/global.
 - /copter_node/Battery: estado de la batería del dron, tomado de /mavros/battery.
 - /copter_node/State: copter_node publica aquí los mensajes recogidos en el topic /mavros/state, que contienen información sobre la aeronave.
 - /copter_node/Velocity: velocidad del dron con respecto al sistema de referencia local a la aeronave, proveniente del topic /mavros/local_position/velocity_body.
- Suscripción:
 - /copter_node/SetVelocity: mediante este topic pueden especificarse consignas de velocidad al dron utilizando mensajes del tipo “TwistStamped”. Sin embargo, el UAV no comenzará a moverse hasta que no se modifique el modo de vuelo a guiado. En la *Figura 48* se observa el funcionamiento de este topic, donde primero se realiza el cambio de modo y posteriormente es enviada la consigna de velocidad. También se muestra la respuesta del dron en una gráfica usando la herramienta *rqt_plot*.

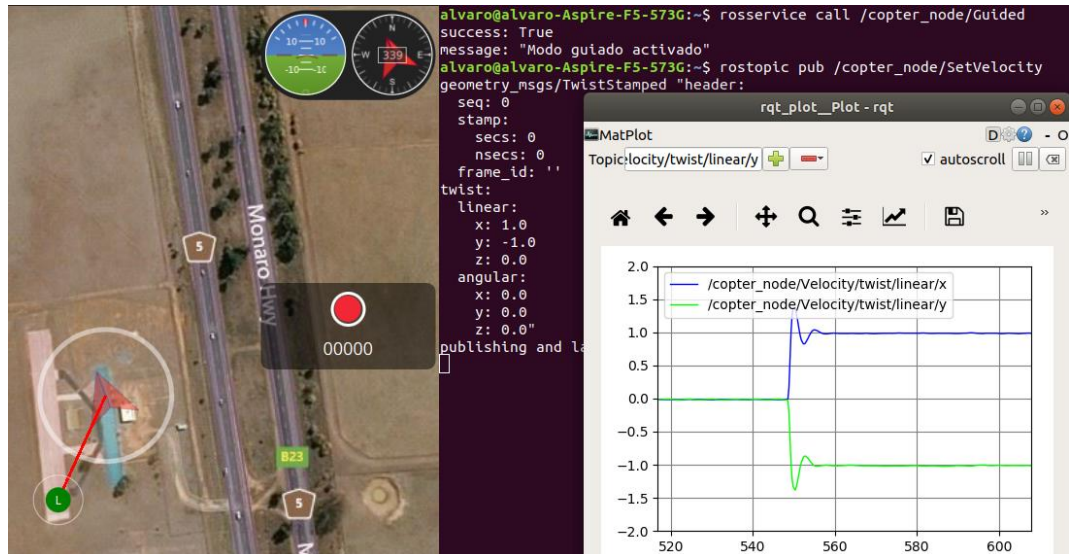


Figura 48: UAV con consigna de velocidad.

- /copter_node/SetPosition: mediante este topic puede especificarse un punto destino en coordenadas locales, utilizando mensajes del tipo "Point". Una vez llegada una consigna por este topic, el dron no solo se dirigirá hasta dicho punto, sino que también se orientará hacia este. Finalmente, al alcanzar la referencia mantendrá esta posición mediante la activación del comando Brake. En la Figura 49 se observa un ejemplo de su funcionamiento.



Figura 49: UAV dirigiéndose al punto x: 100, y: 100, z: 10.

En cuanto a los Servicios:

- /copter_node/Takeoff: este servicio automatiza la rutina de despegue del UAV. Para ello se ha definido el tipo de servicio "takeoff" (Figura 50) que es parecido al "Trigger" de la dependencia "std_srv", pero añadiendo un argumento correspondiente a la altura a la que se quiere despegar. Como resultado la función devuelve un booleano que especifica el éxito de la orden y un mensaje. Al llamar a este servicio el dron comenzará a elevarse hasta la altura especificada y luego mantendrá esa posición activando el modo Brake. En la Figura 51 puede observarse un ejemplo de su funcionamiento.

```
float32 altitude
---
bool success
string message
```

Figura 50: *takeoff.srv*

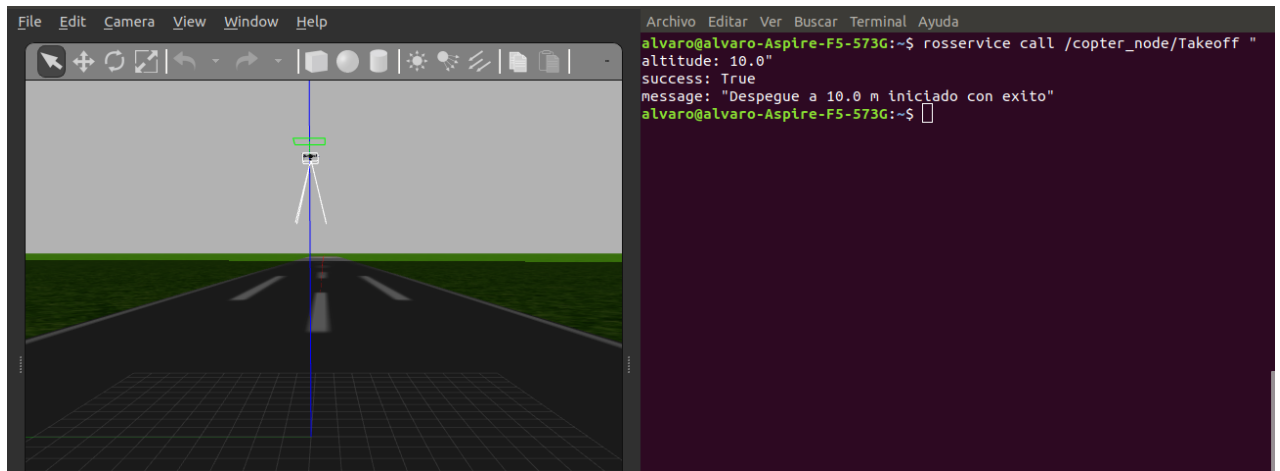


Figura 51: Despegue del UAV a 10 m utilizando el servicio */copter_node/Takeoff*.

- */copter_node/Land*: este servicio cambia el modo de vuelo a Land. Utiliza para ello el tipo “std_srv/Trigger” de la librería estándar de ROS, el cual no tiene argumento de entrada y como respuesta devuelve un booleano con el éxito del comando, y un mensaje.
- */copter_node/RTL*: este servicio primero cambia el punto base actual a la posición inicial del dron y luego activa el modo RTL, tal y como se muestra en la Figura 52. Como en el caso anterior, es un servicio del tipo “std_srv/Trigger”.



Figura 52: Ejemplo de uso del servicio */copter_node/RTL*.

- */copter_node/Stop*: permite cambia el modo de vuelo a Brake, frenando el UAV. Este servicio es también del tipo “std_srv/Trigger”.
- */copter_node/Guided*: cambia el modo de vuelo a guiado, permitiendo el posterior control en velocidad del dron mediante el topic */copter_node/SetVelocity*. Como en los casos anteriores, es del tipo “std_srv/Trigger”.

4.4.1.1 Elaboración del Código

En este apartado se detalla el código desarrollado, así como los diferentes problemas surgidos durante su elaboración. Para una revisión completa del programa véase el *anexo A.1*. En primer lugar, el código comienza con la siguiente línea:

```
#!/usr/bin/env Python
```

Esta línea es necesaria añadirla al inicio de todo nodo ROS escrito en Python y asegura que el programa se ejecute como un script de Python. A continuación, se incluyen las diferentes librerías, definiciones de mensajes y tipos de servicios necesarios para el funcionamiento del código. Ya en el núcleo del programa, se ha definido una clase con el nombre “Copter_node” que contiene el funcionamiento completo del nodo. Más tarde, se realiza una instancia de esta clase dentro de la típica comprobación de la variable `__name__`, como se observa en el siguiente fragmento:

```
if __name__ == "__main__":  
    Copter_node()
```

Dentro del constructor de la clase Copter_node (función `__init__`) se inicializa el nodo, se declaran algunas variables y se anuncian tanto los topics donde el nodo va a publicar y suscribirse como los servicios que ofrece y de cuales va a ser cliente. Por último, se realiza una llamada al servicio `/mavros/setpoint_velocity/mav_frame`, cambiando el sistema de referencia de las consignas de velocidad como se comentó en el *apartado 4.3*, y se concluye con una llamada a la función bucle.

Realizando varias pruebas con el simulador, se ha observado que al aterrizar el UAV en cualquier punto y luego volver a despegar, el punto base o de despegue cambia a esa nueva posición. Esto puede ser un problema si el objetivo es que el dron vuelva al punto inicial, ya que entonces el modo RTL no serviría. Con el objeto de modificar este comportamiento, se guarda la primera posición global que recibe “copter_node” en la variable “initialPoint”, tal y como se muestra en el siguiente fragmento de programa, para más adelante fijar este punto como base siempre que llegue una consigna de posición o el servicio `/copter_node/RTL` sea llamado. A esta posición además se le añade un valor de offset en altitud, para compensar el propio offset del topic `/mavros/home_position/set`. En la *Figura 53* se observa muy bien este fenómeno, donde después de cambiar el punto base sin añadir el valor de offset, el indicador de altura marca que el UAV está a una determinada altitud distinta de cero cuando en realidad se encuentra en el suelo.

```
def globalPosition_callback(self,msg):  
    if self.not_inicialPoint:  
        #registro del punto inicial  
        self.initialPoint = HomePosition()  
        self.initialPoint.geo.latitude = msg.latitude  
        self.initialPoint.geo.longitude = msg.longitude  
        self.initialPoint.geo.altitude = msg.altitude - 19.4  
        self.not_inicialPoint = False  
    self.pub_globalposition.publish(msg)
```



Figura 53: Cambio de punto base mediante `/mavros/home_position/set` sin añadir offset.

Algo que no tiene en cuenta el simulador, pero que es importante a la hora de utilizar cualquier tipo de UAV, es el problema del viento. Este fenómeno meteorológico puede hacer que un dron no pueda mantener su posición fija en un punto. Afortunadamente Ardupilot implementa el modo de vuelo Loiter, en el cual la aeronave trata de mantener automáticamente la posición. Sin embargo, a la hora de probarlo en el simulador se encontró con el problema de que el dron comenzaba a descender. Por ello finalmente se ha utilizado el modo Brake, el cual no solo intenta frenar el UAV lo antes posible, sino que para ello utiliza controladores del tipo Loiter, tal y como comenta el equipo de Ardupilot en su página web [23].

En el siguiente fragmento puede observarse la función bucle. Esta realiza un ciclo infinito que ejecuta diversas acciones. En primer lugar, comprueba si hay una consigna de velocidad, y si es así, publica esta en el topic `/mavros/setpoint_velocity/cmd_vel`. En caso contrario comprueba si el UAV tiene una consigna de posición y además la distancia hasta el punto destino es inferior a 40 cm. Si es cierto se considera que la aeronave ha llegado correctamente al destino y, por tanto, para que mantenga esa posición, cambia el modo de vuelo a Brake. Por último, envía la velocidad local de la aeronave por el topic `/copter_node/Velocity`. El motivo por el que esta acción es realizada en el bucle y no en la respectiva función callback del topic `/mavros/local_position/velocity_body`, es debido a que para el posterior modelado es necesario obtener la velocidad de salida del dron con la misma frecuencia de envío de consignas.

```
def bucle(self):
    while not rospy.is_shutdown():
        if self.Send_Velocity:
            self.pub_setVelocity.publish(self.Vel_msg)
        elif self.reach_Point and
            (self.distance(self.localPosition, self.Goal_Point) < 0.4):
            self.reach_Point = False
            self.change_Mode("brake")
            self.pub_velocity.publish(self.velocity)
            self.rate.sleep()
```

Para elegir la frecuencia de repetición del bucle debe tenerse en cuenta que debe ser igual o inferior a la frecuencia a la que MAVROS publica los mensajes de velocidad. Tal y como se observa en la Figura 54 esta es de unos 4 Hz no muy estables.

Topic	Type	Bandwidth	Hz
▸ <input checked="" type="checkbox"/> /mavros/local_position/velocity_body	geometry_msgs/TwistStamped	274.06B/s	3.76

Figura 54: /mavros/local_position/velocity_body con la herramienta topic monitor de rqt.

Esta frecuencia se debe modificar por una mayor debido a que es demasiado baja y podría afectar a la posterior etapa de modelado. Para este proyecto se ha elegido una nueva frecuencia de 30 HZ. Para variar esta frecuencia se debe ejecutar el siguiente comando en la terminal de MAVPROXY:

```
set streamrate 30
```

En la Figura 55 se comprueba como el cambio de frecuencia se ha realizado con éxito. Por ello se ha elegido precisamente esta frecuencia (30 Hz) como la de repetición del bucle.

Topic	Type	Bandwidth	Hz
▸ <input checked="" type="checkbox"/> /mavros/local_position/velocity_body	geometry_msgs/TwistStamped	2.27KB/s	30.32

Figura 55: Nueva frecuencia de envío de mensajes

En el siguiente fragmento puede observarse la función que maneja el servicio /copter_node/Takeoff. Esta comienza desactivando el envío de velocidades para evitar errores en el despegue. Imaginemos que antes de despegar el dron, se enviase una consigna de velocidad por el topic /copter_node/SetVelocity. El UAV no se movería del sitio porque no está armado, sin embargo, estaría recibiendo esa consigna continuamente. Ahora si se iniciase el despegue, la anterior consigna de velocidad afectaría, de tal manera que incluso fuera imposible despegar el dron (por ejemplo, con una consigna de velocidad cero). A continuación, inicia la misma secuencia que se comentó en el apartado 4.1.1.1 para el despegue: primero un cambio al modo guiado, luego armado de los motores utilizando el servicio /mavros/cmd/arming y por último llamada al servicio /mavros/cmd/takeoff (con todos sus argumentos a 0 menos la altura) para el despegue. Si todo va bien, se especifica la posición actual a la altura de despegue introducida como el nuevo punto a alcanzar, para que cuando termine el despegue se active el modo Brake.

```
def takeoff_function(self, req):
    self.Send_Velocity = False
    if self.change_Mode("guided"): #cambio a modo guiado
        Armado = self.Arming_service(True) #Armado de los motores
        if Armado.success:
            Despegue = self.takeoff_service(0,0,0,0, req.altitude) #despegue
            if Despegue.success:
                self.reach_Point = True
                self.Goal_Point =
                Point(self.localPosition.x, self.localPosition.y, req.altitude)
                return takeoffResponse(True, "Despegue a " + str(req.altitude)
                + " m iniciado con éxito")
            return takeoffResponse(False, "Error al iniciar el despegue")
        else:
            return takeoffResponse(False, "Error en el armado de los motores")
    else:
        return takeoffResponse(False, "Error, no se ha podido cambiar el modo
        de vuelo a guiado")
```

En el siguiente fragmento puede observarse la función callback del topic /copter_node/SetVelocity. Su funcionamiento es simple: activar el envío de mensajes de

velocidad y guardar la consigna en la variable `vel_msg` para más adelante enviar esta de forma repetitiva al UAV en la función bucle.

```
def setVelocity_callback(self,msg):
    self.Send_Velocity = True
    self.reach_Point = False
    self.Vel_msg = msg
    rospy.loginfo("consigna de velocidad: " + str(msg.twist))
```

En cuanto al topic `/copter_node/SetPosition`, comienza con una sentencia `if` donde comprueba si el UAV ya está siguiendo alguna consigna de velocidad. Si es así, no realiza ninguna acción, de esta manera `copter_node` implementa una mayor prioridad a las consignas de velocidad frente a las de posición. El motivo por el que se ha especificado esta preferencia es pensando en que las consignas de posición podrían utilizarse para dirigir al UAV por un camino, mientras que las de velocidad para esquivar obstáculos. Si por el contrario no hubiera consignas de velocidad, primero cambiaría el punto base actual por el inicial, luego activaría el modo guiado, calcularía el ángulo de guiñada para orientar el dron al punto destino, transformaría esta orientación a cuaternios y publicaría la pose destino en el topic `/mavros/setpoint_position/local`.

4.4.2 Mission_node

`Mission_node` (Anexo A.2) implementa la posibilidad de trabajar con misiones definidas mediante ficheros. Este nodo no está suscrito ni publica en ningún topic. Por el contrario, su funcionamiento se basa en dos servicios: `/mission_node/upload` y `/mission_node/startmission`.

4.4.2.1 /mission_node/upload

El servicio `/mission_node/upload` permite cargar al UAV misiones definidas en ficheros. Para este servicio se ha definido el tipo `"fileMission"` (Figura 56), el cual es parecido al tipo `"Trigger"`, pero con una cadena de texto como argumento de entrada, correspondiente a la ruta en la que se encuentra el archivo.

```
string filePath
---
bool success
string message
```

Figura 56: `fileMission.srv`

En la Figura 57 se observa un ejemplo de trayectoria cuadrada definida desde fichero, donde cada fila representa un punto de la trayectoria (el cuarto vértice es el punto inicial). Cada punto debe contener los siguientes parámetros en orden y separados mediante tabulaciones: latitud, longitud, altura, tiempo que permanece esperando el UAV en dicho punto y velocidad con la que se dirige a este. La posición de cada punto debe expresarse en coordenadas globales con el parámetro de altura relativo al punto base. En concreto para definir esta trayectoria de ejemplo, primero se ha movido el dron a cada uno de los vértices del cuadrado y luego se han apuntado estas posiciones. Por otro lado, tanto el delay como la velocidad deben ser números

positivos. Se recomienda para la generación de trayectorias, completar una plantilla en Excel y luego exportar el resultado a txt. Por ello se ha elegido este formato con tabulaciones, debido a que es la forma en la cual Excel exporta los datos a fichero de texto.

#latitud	#longitud	#altura	#delay	#vel
-35.362395	149.165232	10	0	5
-35.362382	149.166339	10	10	2
-35.363269	149.166353	10	0	1

Figura 57: Fichero de definición de trayectoria cuadrada.

Finalmente, en la Figura 58 se observa un ejemplo de empleo de este servicio, así como los diferentes errores que pueden darse en su uso.

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice call /mission_node/upload "filePath:
'/home/alvaro/Escritorio/cuadrado.txt'"
success: True
message: "Mission uploaded"
alvaro@alvaro-Aspire-F5-573G:~$
```

a) Mission cargada con éxito.

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice call /mission_node/upload "filePath:
'/home/alvaro/Escritorio/CUADRADO.txt'"
success: False
message: "Error: la ruta introducida es incorrecta o algun campo esta incompleto"
alvaro@alvaro-Aspire-F5-573G:~$
```

b) Error en la ruta.

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice call /mission_node/upload "filePath:
'/home/alvaro/Escritorio/cuadrado.txt'"
success: False
message: "Error: Algun campo esta incorrecto, revise el archivo"
alvaro@alvaro-Aspire-F5-573G:~$
```

#latitud	#longitud	#altura	#delay	#vel
-35.362395	149.165232	10	0	-5
-35.362382	149.166339	10	10	2
-35.363269	149.166353	10	0	2

c) Campo incorrecto (velocidad debe ser mayor que 0).

```
alvaro@alvaro-Aspire-F5-573G:~$ rosservice call /mission_node/upload "filePath:
'/home/alvaro/Escritorio/cuadrado.txt'"
success: False
message: "Error: la ruta introducida es incorrecta o algun campo esta incompleto"
alvaro@alvaro-Aspire-F5-573G:~$
```

#latitud	#longitud	#altura	#delay	#vel
-35.362395	149.165232	10	0	
-35.362382	149.166339	10	10	2
-35.363269	149.166353	10	0	2

d) Campo incompleto.

Figura 58: Ejemplo de uso del servicio /mission_node/upload.

4.4.2.2 /mission_node/startMission

Por otro lado, el servicio `/mission_node/startMission` permite comenzar la misión que esté cargada actualmente en el UAV. Para ello inicialmente ejecuta un despegue a 10 m y después de un cierto tiempo realiza un cambio al modo de vuelo auto. Para este servicio se ha utilizado el tipo “`std_srv/Trigger`”, ya comentado en el apartado anterior. En la *Figura 59* puede observarse un ejemplo de uso de este servicio, en el cual el UAV sigue la trayectoria cuadrada antes especificada.



Figura 59: UAV siguiendo trayectoria cuadrada después de llamar al servicio `/mission_node/startMission`.

4.4.2.3 Elaboración del código

4.4.2.3.1 Pruebas iniciales

Antes de comenzar a definir el nodo, fue necesario realizar una serie de pruebas para familiarizarse con los diferentes comandos y sus parámetros. Para ello se ha utilizado el código “`Prueba_wp.py`” (*Anexo A.3*). Las misiones en MAVROS se definen mediante listas de mensajes del tipo “`mavros_msg/Waypoint`”. Estos mensajes, como se observa en la *Figura 60*, están compuestos por: un campo *frame* para definir el sistema de referencia, *command* para especificar el tipo de comando MAVLink a ejecutar y una serie de 7 parámetros (entre ellos *x_lat*, *y_long* y *z_alt* como los parámetros 5, 6 y 7 respectivamente) cuyo significado depende del comando introducido.

```
alvaro@alvaro-Aspire-F5-573G:~$ rosmmsg show mavros_msgs/Waypoint
uint8 FRAME_GLOBAL=0
uint8 FRAME_LOCAL_NED=1
uint8 FRAME_MISSION=2
uint8 FRAME_GLOBAL_REL_ALT=3
uint8 FRAME_LOCAL_ENU=4
uint8 frame
uint16 command
bool is_current
bool autocontinue
float32 param1
float32 param2
float32 param3
float32 param4
float64 x_lat
float64 y_long
float64 z_alt

alvaro@alvaro-Aspire-F5-573G:~$
```

Figura 60: Contenido de un mensaje de tipo `mavros_msgs/Waypoint`.

Existen una gran cantidad de comandos MAVLink y puede encontrarse documentación sobre cada uno de estos en su página web [24]. Para este proyecto se han considerado interesantes los siguientes:

- `MAV_CMD_NAV_WAYPOINT` (16): navegación hacia un waypoint. En orden, cada uno de sus parámetros significa: tiempo que permanece el UAV en dicho punto, distancia a la que se considera alcanzado el punto, radio de paso, ángulo de guiñada y por último posición del punto. Este comando no permite utilizar otro sistema de referencia distinto a `FRAME_GLOBAL_REL_ALT` (coordenadas globales con altura relativa al punto de lanzamiento). Es por ello que en el fichero se utiliza este sistema de referencia para definir la trayectoria.
- `MAV_CMD_NAV_RETURN_TO_LAUNCH` (20): vuelta al punto de lanzamiento (RTL), no requiere ningún parámetro.
- `MAV_CMD_NAV_TAKEOFF` (22): despegue del UAV. En orden, los parámetros expresan: ángulo de cabeceo, tanto el segundo como el tercero no son utilizados, ángulo de guiñada y Posición.
- `MAV_CMD_DO_CHANGE_SPEED` (178): permite cambiar la velocidad del UAV y el porcentaje de aceleración. Los parámetros en orden corresponden a: tipo de velocidad (terrestre, aérea, de subida o de bajada), velocidad, porcentaje de aceleración donde -1 significa sin cambios y si esa velocidad es absoluta (0) o relativa (1). Los parámetros restantes (quinto, sexto y séptimo) no se utilizan.

Durante las pruebas realizadas con estos comandos, surgieron dos principales problemas. En primer lugar, no se ha podido realizar un despegue del UAV mediante el comando correspondiente. Se experimentaron distintas opciones: cambiando directamente a modo auto, armando los motores primero y luego usando el modo auto, incluso se probó a iniciar la misión desde QGroundControl pero lo único que se consiguió fue el error que aparece en la *Figura 61*. También se utilizó el comando `start` de MAVLink que allí se menciona sin mucho éxito.

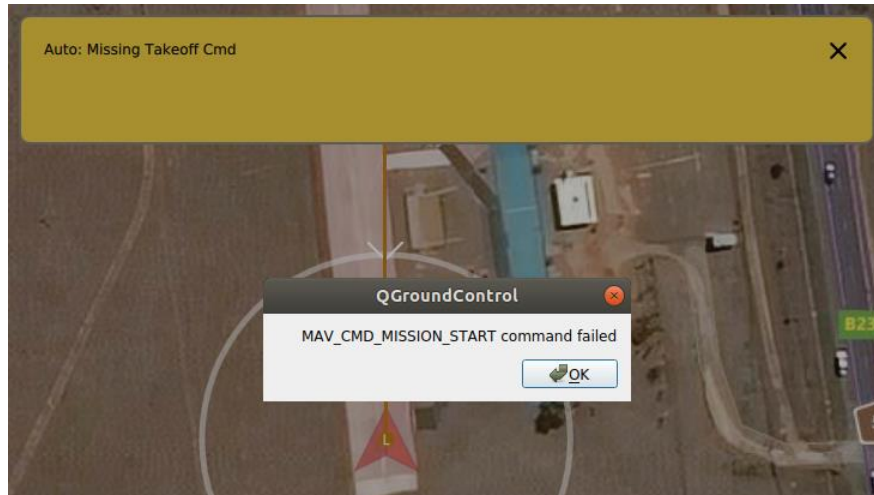


Figura 61: Error al iniciar misión con comando takeoff en QGroundControl.

Finalmente se planteó como solución a este problema no utilizar este comando y en su lugar usar el servicio `/copter_node/Takeoff` para el despegue e iniciar la misión una vez el UAV ya estuviera en el aire. Fue ahí cuando se presentó el segundo problema. Resulta que Ardupilot entiende que el primer comando de la lista de waypoints corresponde siempre al despegue. Por ello al iniciar una misión desde el aire ignora siempre el primer elemento de la lista. Como solución a este problema se ha añadido un comando arbitrario al inicio de la lista. De esta manera se obtiene la certeza de que el comando ignorado no pertenece a la trayectoria deseada.

4.4.2.3.2 Código en detalle

Para el desarrollo del nodo `mission_node`, se ha seguido la misma estructura que en el caso de `copter_node`. Primero son incluidas las dependencias necesarias, luego se define la clase `Mission_node` (que contiene el funcionamiento completo del nodo) y por último se realiza una instancia de esa clase dentro de la comprobación del parámetro `__name__` de Python. En la función `__init__` de la clase, se inicializa el nodo y se anuncian los dos servicios que ofrece y de cuáles será cliente, finalmente se realiza una llamada a la función `spin` de ROS.

El servicio `/mission_node/upload` (`upload_function`) ejecuta las siguientes acciones: en primer lugar, realiza una llamada a la función de procesamiento de datos, `read_file`, con la ruta del fichero como argumento. Si todo está correcto llama a la función `/mavros/mission/clear` para borrar la misión actual y si todo va bien, realiza otra llamada al servicio `/mavros/mission/push` con la lista de waypoints devuelta por `read_file`.

La función `read_file` puede dividirse en dos partes: una de lectura de fichero y otra de conversión de datos a waypoints de MAVROS. La función comienza con una apertura del archivo en modo lectura y luego una llamada al método `readlines`, que devuelve una lista donde cada elemento corresponde a una fila del documento. A continuación, dentro de un bucle `for`, extrae uno a uno los datos y los añade a la matriz “data”. Ya en la segunda parte, genera la lista “misión”, con un primer comando arbitrario (que será ignorado) y agrega por cada fila de la matriz “data” un comando de cambio de velocidad y uno de navegación hacia un waypoint. Una vez terminado agrega un último comando `RTL` y retorna la lista generada.

4.5 Modelado

4.5.1 Toma de datos

Para poder obtener el modelo del dron, antes es necesario determinar cuál es su respuesta ante entradas conocidas. Para ello se ha utilizado el nodo Modelado (*Anexo A.4*). Este programa comienza con el despegue del UAV y después de un tiempo envía un tren de pulsos como consigna de velocidad a cada uno de los tres ejes de movimiento y al giro en Z. Cada uno de estos pulsos tienen una duración de 30 segundos, tiempo suficiente para asegurar que el dron alcanza el valor de referencia. Es importante también destacar el valor de velocidad de cada uno de estos pulsos ya que el modelo que se obtenga finalmente solo será válido para un entorno cercano a esa velocidad. Por ello antes de realizar el modelo hay que preguntarse cuál será la velocidad de operación de la aeronave. Para este proyecto se ha realizado el modelo a una velocidad lineal de 1m/s y angular de 1 rad/s, aunque si se quisiese cambiar estas velocidades se podrían fácilmente modificar las variables “VelLinear” y “VelAngular”. Finalmente, en la *Figura 62* puede observarse un ejemplo del nodo Modelado en funcionamiento, donde además se representa en una gráfica la velocidad de salida de la aeronave usando la herramienta *rqt_plot*.

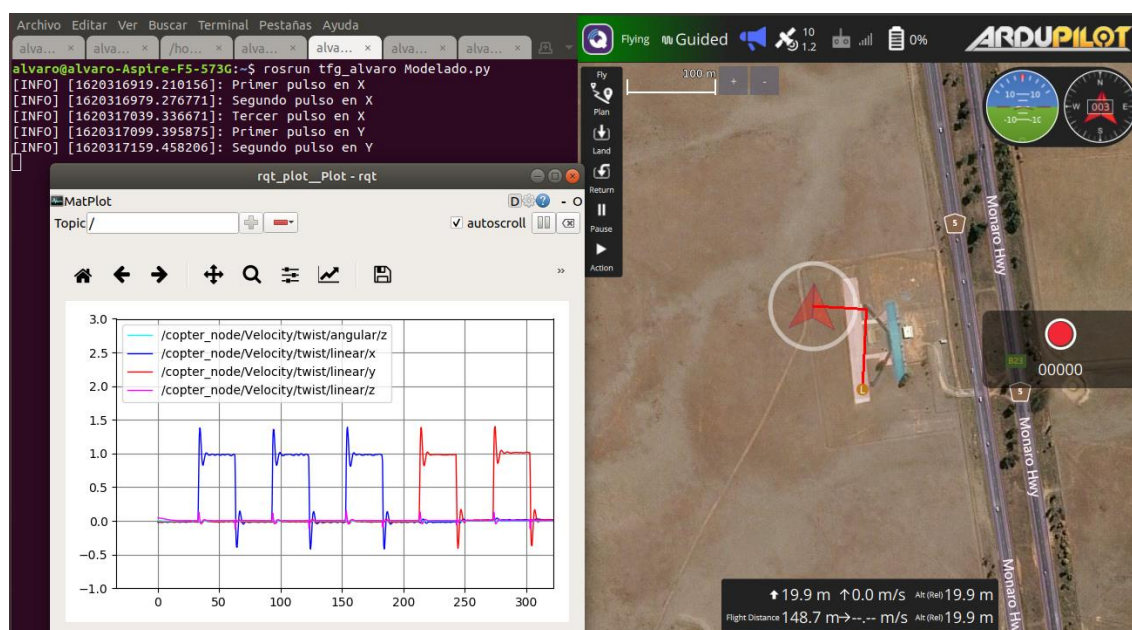


Figura 62: Ejecución del nodo Modelado.

No obstante, antes de ejecutar el nodo Modelado y enviar las referencias, primero se inició la grabación de los topics de velocidad tanto de entrada como de salida mediante la introducción del siguiente comando en una terminal:

```
roslaunch record /copter_node/Velocity /mavros/setpoint_velocity/cmd_vel
```

Una vez terminada la ejecución del nodo, se detuvo la grabación obteniendo como resultado un fichero de extensión “bag”. Estos datos son convertidos en dos ficheros csv (*Figura 63*), uno para los datos de entrada (in.csv) y otro para los de salida (out.csv), para facilitar su posterior estudio en Matlab.

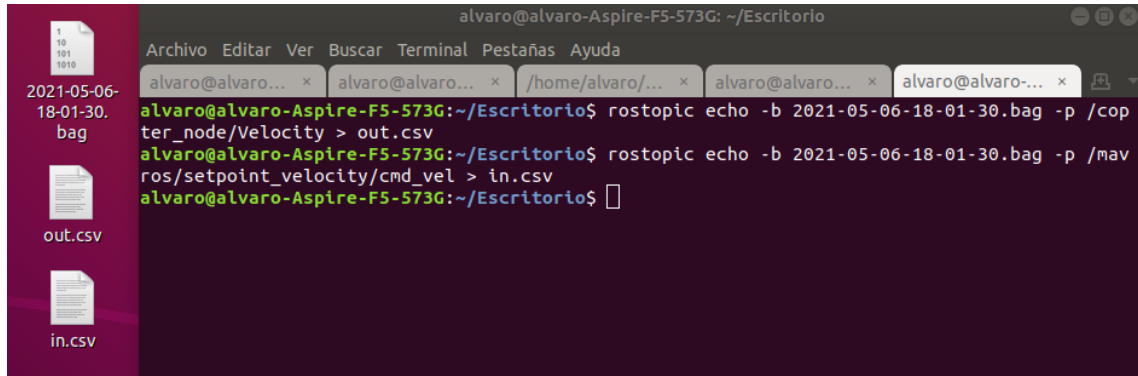


Figura 63: Conversión de datos en archivo bag a fichero csv.

4.5.2 Estimación del modelo en Matlab

Una vez recogidos los datos, el siguiente paso es estimar los parámetros de primer orden del modelo cinemático. Para ello se ha hecho uso de la herramienta *System Identification* de Matlab. Sin embargo, al importar los datos a Matlab se encontró con el problema de que había más datos de salida que de entrada, tal y como se observa en la *Figura 64*. Esto se produce debido a que durante el tiempo que transcurre desde que comienza la grabación hasta que se especifica la primera consigna de velocidad, se reciben datos de salida del UAV. Para solucionar este inconveniente se desarrolló el script *medidas.m* (*Anexo A.5*), que adicionalmente separa cada una de las velocidades en distintas variables además de representarlas gráficamente (*Figura 65*).

Workspace	
Name	Value
in	22067x10 table
out	22877x10 table

Figura 64: Diferencia en cantidad de datos de entrada y salida.

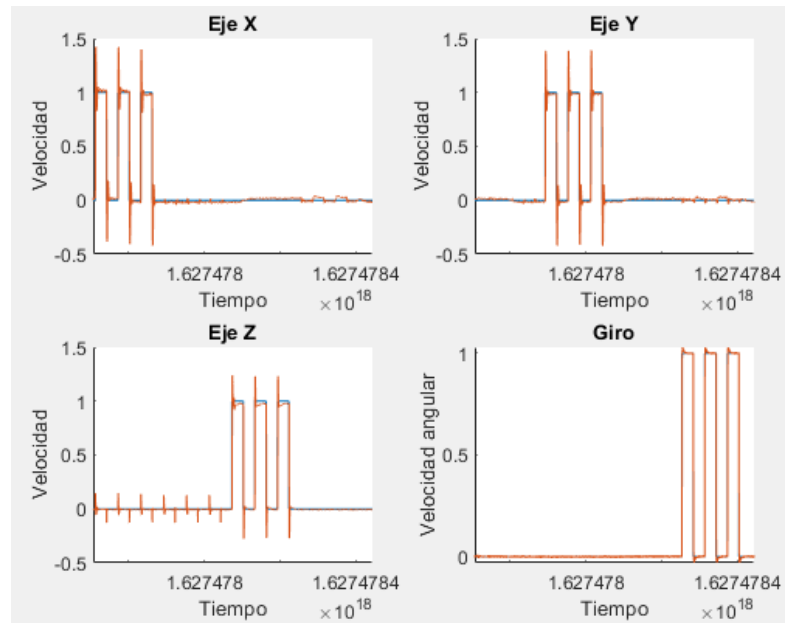


Figura 65: Visualización de los datos en gráficas.

Ya acondicionado los datos, se importaron en system identification seleccionando *import data* -> *time domain data* y rellenando los parámetros como se muestra en el ejemplo de la Figura 66 para el eje X. Quizás uno de los parámetros más destacables puede ser el tiempo de muestreo. Este valor es igual a la inversa de la frecuencia de repetición del bucle de `copter_node`, que como se comentó en el Apartado 4.4.1.1 es de 30 Hz.

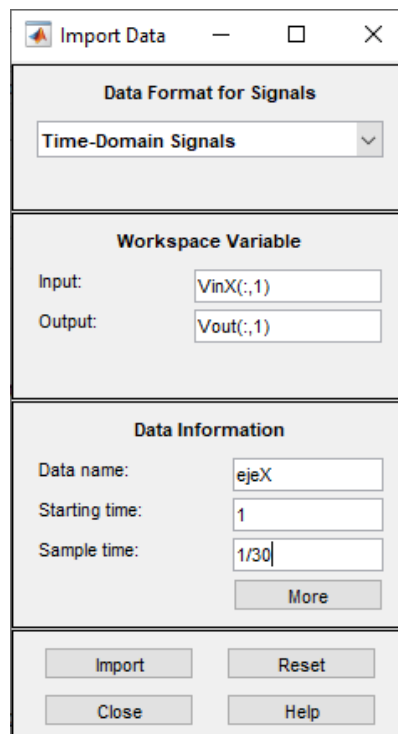


Figura 66: Importar datos a System Identification (eje X).

Con los datos en *System Indetification*, y con el objeto de mejorar la estimación, se selecciona para cada eje el rango de datos donde actúan sus respectivas consignas mediante la

opción *Preprocess* -> *Select Range*. En la Figura 67 se observa un ejemplo donde se restringe el rango de datos para el caso de las velocidades en el eje X.

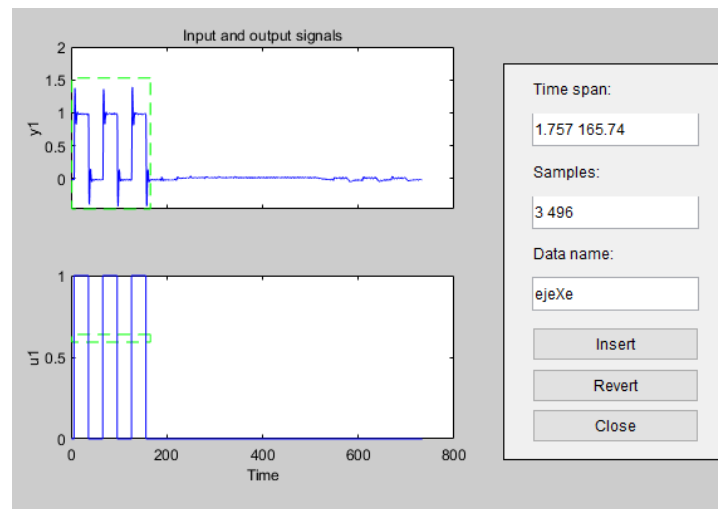


Figura 67: Selección de rango (velocidad eje X).

Por último, para estimar las constantes de primer orden del modelo, se siguieron los siguientes pasos para cada uno de los cuatro ejes:

- Se arrastraron uno a uno los datos con el rango restringido al espacio de trabajo.
- Se seleccionó la opción *Estimate* -> *Process Models*.
- En la ventana emergente, se eligió como parámetros de estimación un modelo con un único polo (primer orden) y se desmarcó la opción de *delay*.

Como resultado se han obtenido los parámetros de la Figura 68.

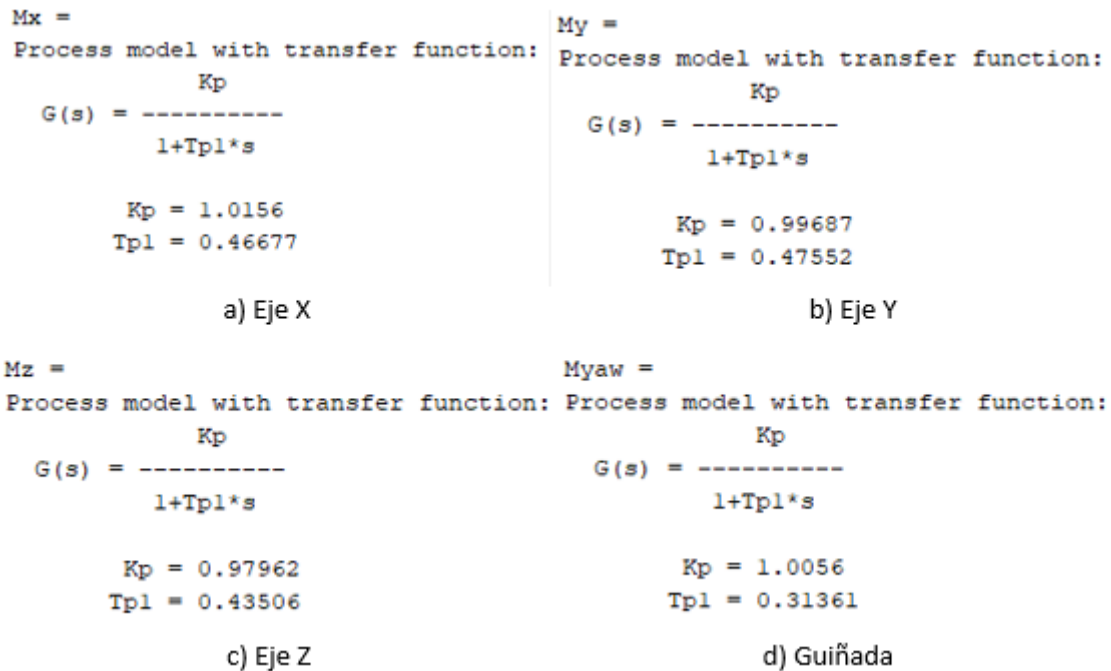


Figura 68: Parámetros modelo de primer orden sin retardo.

En general la ganancia obtenida es cercana a uno, lo que quiere decir que el dron acaba alcanzando satisfactoriamente el valor de entrada u para cada una de las variables de control. En cuanto a las constantes de tiempo, se puede concluir que el sistema tiene una velocidad de respuesta aceptable, sobre todo en los giros. Haciendo uso de la *Ecuación 2* se puede calcular el tiempo de establecimiento para cada eje, obteniendo de media 1,88 segundos para los ejes X e Y, 1,74 segundos para la velocidad en Z y 1,25 segundos para los giros. A parte de estos parámetros, también se adquiere con las estimaciones un porcentaje de su parecido con la respuesta real del sistema. Los más bajos, como era de esperar debido a las sobreoscilaciones en régimen transitorio de la respuesta real, son los de los ejes X e Y con un 79,73 % y 80 % respectivamente, seguidos por el eje Z con un 86,63 % y finalmente el giro, donde se obtiene la mejor estimación, con un 95,06 %.

4.5.3 Comparación de funciones estimadas con valores reales

En este apartado se realiza un estudio comparativo entre el modelo obtenido del dron y los datos reales. En primer lugar, puede observarse en la *Figura 69* la señal real de velocidad del eje X en negro y su modelo en violeta. En general el modelo se ajusta debidamente al comportamiento del dron salvo en el último pulso, donde hay un ligero error entre el modelo y la señal real en régimen permanente debido a que el dron no consigue llegar correctamente a la referencia marcada. Esta conducta extraña refuerza más la necesidad de utilizar varios pulsos para la realización del modelado, disminuyendo así los errores en la estimación. A parte, las dos curvas tienen algunas ligeras diferencias debido a efectos que no se han tenido en cuenta. Entre ellos se encuentran el nivel de ruido de los valores reales y las sobre oscilaciones en régimen transitorio, producidas por efectos de segundo orden. Estas últimas son bastante grandes (cercanas al 40%) pero breves. Adicionalmente, se distinguen una serie de perturbaciones en la parte final del gráfico correspondientes con los tres pulsos en el eje de giro, por lo que puede decirse que los movimientos de guiñada afectan también al eje X de movimiento.

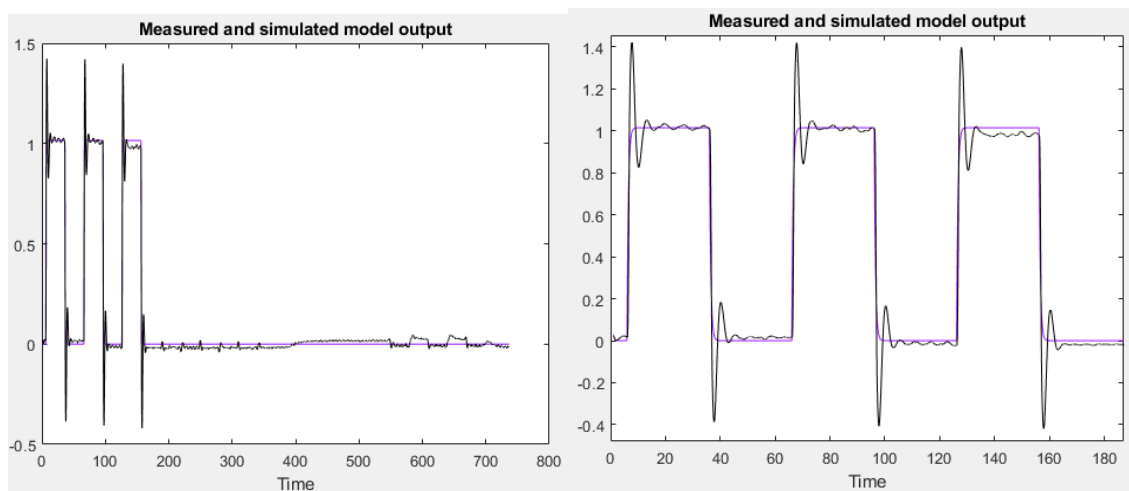


Figura 69: Señal real eje X Vs Modelo de primer orden con retardo.

En la *Figura 70*, puede observarse el comportamiento real del dron en el eje Y (negro) junto con su modelo (azul). Este eje tiene un comportamiento bastante parecido al del anterior, teniendo también sobre oscilaciones altas y breves en régimen transitorio además de perturbaciones en los movimientos de guiñada.

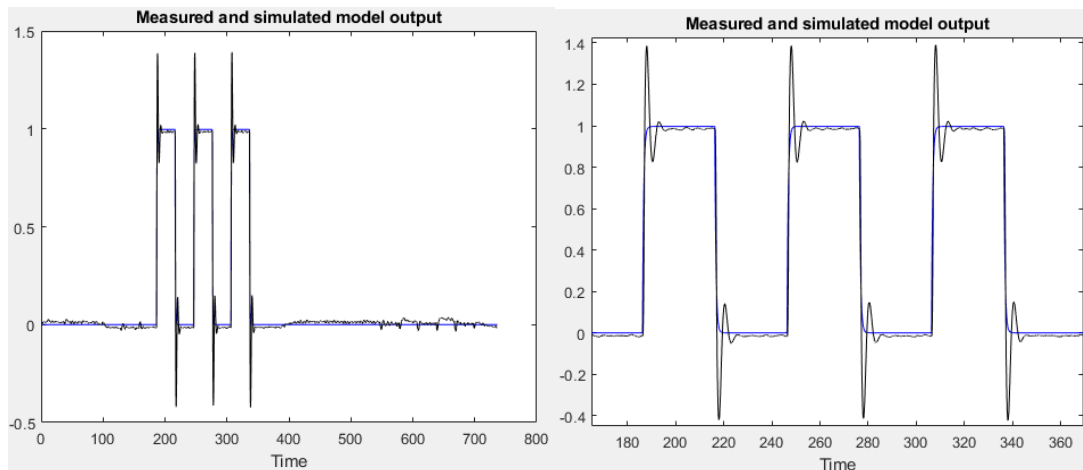


Figura 70: Señal real eje Y Vs Modelo de primer orden con retardo.

En la Figura 71, se muestra esta vez el comportamiento real del dron en el eje Z (negro) junto con su respectivo modelo (azul). En este caso se observa que el ruido es casi imperceptible, y que, aunque si se producen sobre oscilaciones éstas son más pequeñas. Al mismo tiempo se aprecia como este eje no está siendo afectado por movimientos de guiñada, aunque la sucesión de picos antes de la llegada de su respectiva consigna parece indicar que si puede ser influenciado por movimientos bruscos en el plano XY.

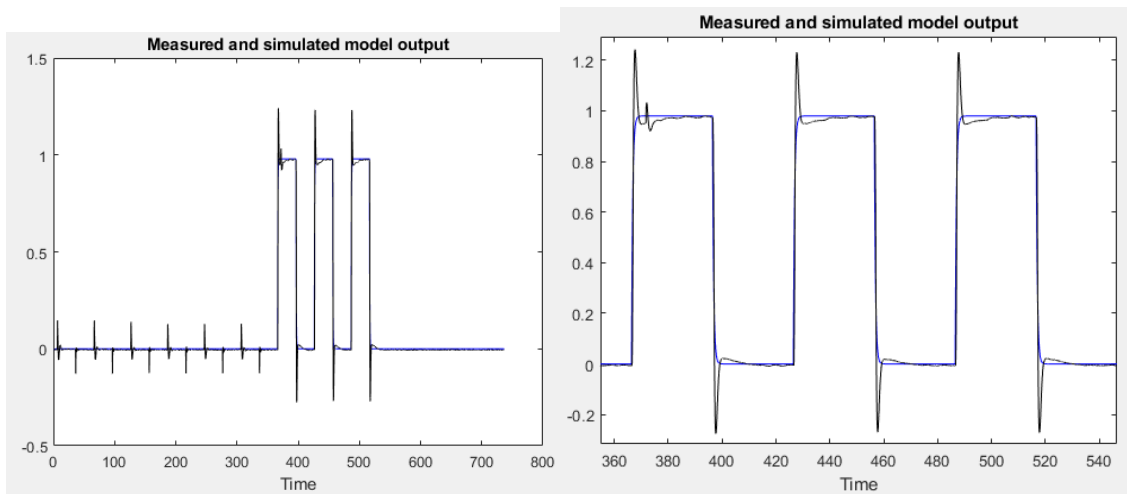


Figura 71: Señal real eje Z Vs Modelo de primer orden con retardo.

Por último, la Figura 72 muestra el comportamiento de guiñada. Como en el caso anterior el ruido es imperceptible y aunque presente sobre oscilaciones en régimen transitorio, lo cierto es que son tan pequeñas que pueden despreciarse. Ello unido con que parece no estar afectado por los movimientos de los demás ejes es una de las razones por las que este eje tiene un porcentaje de estimación tan alto.

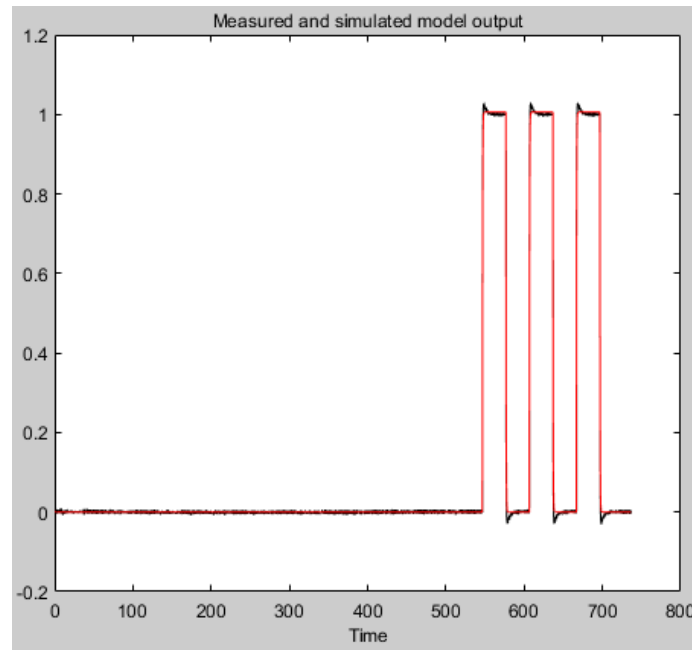


Figura 72: Señal real eje de giro Vs Modelo de primer orden con retardo.

Sintetizando puede decirse que en general los modelos obtenidos reflejan bastante bien el comportamiento del dron en cada uno de los ejes principales salvo en algunas situaciones producidas por efectos de segundo orden. Estas en gran parte están producidas por cambios bruscos en velocidad, por lo que para una buena utilización del modelo se aconseja el uso de polinomios de suavizado de trayectorias que atenúen estos comportamientos.

5. Conclusión

El trabajo realizado en este proyecto puede dividirse en tres partes: estudio del entorno de simulación, desarrollo del nodo interfaz y modelado del dron. En la primera parte, se destaca el uso del software QGroundControl. Su intuitiva interfaz compuesta por botones y sliders ha permitido aprender fácilmente los conceptos básicos de control de drones. Por ello se recomienda su uso a todo aquel que trabaje por primera vez con drones. También se ha realizado un estudio de los topics y servicios de MAVROS, destacando en esta memoria los más interesantes para el desarrollo del nodo interfaz.

En cuanto a la interfaz, su desarrollo no estuvo exento de problemas, no obstante, el resultado final es el esperado. Su funcionamiento se ha dividido en dos nodos distintos: uno para el control y monitorización del UAV y otro únicamente para el uso de misiones definidas mediante fichero.

Finalmente se ha podido realizar un modelado de primer orden del dron simulado. Tomando los datos de velocidad de salida frente a determinadas entradas y estimando posteriormente estos comportamientos con funciones de primer orden mediante la herramienta *System identification* de Matlab.

Como norma general se ha conseguido cumplir todos los objetivos fijados de forma satisfactoria. Este proyecto me ha permitido adentrarme en el mundo de los drones, aprendiendo sobre uno de los softwares autopiloto de código abierto más utilizados, Ardupilot, y todo ello sobre un entorno de simulación 3D.

6. Trabajos Futuros

Como ampliación a este proyecto se proponen los siguientes trabajos:

- Desarrollo de un software de control del UAV partiendo del modelo adquirido en este proyecto.
- Elaboración de un modelo más complejo de segundo orden.
- Modelaje del dron real del departamento.

7. Contenido del CD

Se hace entrega de un CD con los siguientes contenidos:

- Memoria en formato PDF.
- Paquete ROS, tfg_alvaro, con la interfaz desarrollada.
- Plantilla Excel para la creación de misiones.
- Misión de ejemplo cuadrado.txt
- Datos de entrada (in.csv) y salida (out.csv) tomados en el modelado.
- Script desarrollado de Matlab (*Anexo A.5*) y sesión guardada de *System Identification* (indenticacion.sid)

Bibliografía

- [1] R. Assignment, “Modeling an Orange,” pp. 1307–1309.
- [2] M. Bonelli, “Mundo Drone: Historia de los Drones,” *Mundo Drone*. 2016, Accessed: May 07, 2021. [Online]. Available: <http://eldrone.es/historia-de-los-drones/>.
- [3] A. Ollero, *Aerial Robotic Manipulators*. 2019.
- [4] ROS.org, “Sobre ROS,” 2021. <https://www.ros.org/about-ros/> (accessed Mar. 28, 2021).
- [5] “Servidor de parámetros - ROS Wiki.” [http://wiki.ros.org/Parameter Server](http://wiki.ros.org/Parameter%20Server) (accessed Mar. 29, 2021).
- [6] “Pixhawk 4 | Guía del usuario de PX4.” https://docs.px4.io/master/en/flight_controller/pixhawk4.html (accessed Apr. 05, 2021).
- [7] “ArduPilot :: Acerca de.” <https://ardupilot.org/index.php/about> (accessed May 05, 2021).
- [8] ArduPilot Dev Team, “SITL Simulator (Software in the Loop),” *Ardupilot*, 2016. <https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html> (accessed Apr. 01, 2021).
- [9] “Gazebo.” <http://gazebo.org/> (accessed May 04, 2021).
- [10] “Introducción · Guía para desarrolladores de MAVLink.” <https://mavlink.io/en/> (accessed Apr. 04, 2021).
- [11] Vladimir Ermakov, “mavros - ROS Wiki,” *ROS.org*. <http://wiki.ros.org/mavros> (accessed Apr. 04, 2021).
- [12] A. Tridgell, P. Barker, and S. Dade, “MAVProxy - documentación de MAVProxy 1.6.4.” <https://ardupilot.org/mavproxy/> (accessed Apr. 01, 2021).
- [13] QGroundControl, “Firmware · QGroundControl User Guide,” 2017. 2019, Accessed: Apr. 04, 2021. [Online]. Available: <https://docs.qgroundcontrol.com/master/en/index.html>.
- [14] M. Castillo-Lopez, P. Ludivig, S. A. Sajadi-Alamdari, J. L. Sanchez-Lopez, M. A. Olivares-Mendez, and H. Voos, “A Real-Time Approach for Chance-Constrained Motion Planning with Dynamic Obstacles,” *IEEE Robot. Autom. Lett.*, vol. 5, no. 2, pp. 3620–3625, 2020, doi: 10.1109/LRA.2020.2975759.
- [15] “Ubuntu 18.04.5 LTS (Bionic Beaver).” <https://releases.ubuntu.com/18.04/> (accessed Mar. 15, 2021).
- [16] B. A. L. Fm, “Linux : cómo instalarlo junto a Windows 10 para usar ambos en un PC,” 2019, Accessed: Mar. 15, 2021. [Online]. Available: <https://www.xataka.com/basics/como-instalar-linux-a-windows-10-ordenador>.
- [17] ROS, “melodic/Installation/Ubuntu - ROS Wiki,” 2018. <http://wiki.ros.org/melodic/Installation/Ubuntu> (accessed Mar. 23, 2021).
- [18] ArduPilot Dev Team, “Setting up the Build Environment (Linux/Ubuntu) — Dev documentation,” 2018. <https://ardupilot.org/dev/docs/building-setup-linux.html#building-setup-linux> (accessed May 05, 2021).

- [19] “Using Gazebo Simulator with SITL — Dev documentation.”
<https://ardupilot.org/dev/docs/using-gazebo-simulator-with-sitl.html#plugin-installation> (accessed May 04, 2021).
- [20] “mavros/mavros at master · mavlink/mavros · GitHub.”
<https://github.com/mavlink/mavros/tree/master/mavros#installation> (accessed Mar. 31, 2021).
- [21] Dronecode, “Download and Install · QGroundControl User Guide,” *Docs qgroundcontrol*. 2016, Accessed: May 04, 2021. [Online]. Available:
https://docs.qgroundcontrol.com/master/en/getting_started/download_and_install.html#ubuntu.
- [22] “ROS — Dev documentation.” <https://ardupilot.org/dev/docs/ros-sitl.html> (accessed Apr. 29, 2021).
- [23] ArduPilot, “ArduPilot - Copter documentation.”
<https://ardupilot.org/copter/docs/brake-mode.html#brake-mode> (accessed May 16, 2021).
- [24] “Messages (common) · MAVLink Developer Guide,” 2020.
https://mavlink.io/en/messages/common.html#mav_commands (accessed May 18, 2021).

Anexos

A. Códigos

A.1 copter_node.py

```
#!/usr/bin/env python

import rospy
from tf.transformations import quaternion_from_euler
from math import sqrt, atan2
from std_srvs.srv import Trigger, TriggerResponse
from tfg_alvaro.srv import *
from mavros_msgs.srv import SetMode, SetModeResponse, SetModeRequest
from mavros_msgs.srv import CommandBool, CommandBoolResponse, CommandBoolRequest
from mavros_msgs.srv import CommandTOL, CommandTOLResponse, CommandTOLRequest
from mavros_msgs.srv import SetMavFrame, SetMavFrameResponse, SetMavFrameRequest
from geometry_msgs.msg import PoseStamped, TwistStamped, Point
from sensor_msgs.msg import NavSatFix, BatteryState
from mavros_msgs.msg import HomePosition, State

class Copter_node():
    def __init__(self):
        rospy.init_node("copter_node") #inicializar nodo
        #variables
        self.rate = rospy.Rate(30)      #30 Hz
        self.Send_Velocity = False      #variable que representa si se envia o no
        #mensajes de velocidad
        self.reach_Point = False         #variable que representa si el dron debe
        #alcanzar un punto especificado
        self.localPosition = None        #posicion local del dron
        self.Goal_Point = None           #posicion especificada que se pretende
        #alcanzar
        self.not_inicialPoint = True     #variable representa si ya se ha
        #registrado la posicion inicial como casa
        self.velocity = TwistStamped()   #velocidad local del UAV
        #Publicaciones
        self.pub_localposition =
        rospy.Publisher('/copter_node/Local_position', PoseStamped, queue_size=10)
        self.pub_globalposition =
        rospy.Publisher('/copter_node/Global_position', NavSatFix, queue_size=10)
        self.pub_battery =
        rospy.Publisher('/copter_node/Battery', BatteryState, queue_size=10)
        self.pub_velocity =
        rospy.Publisher('/copter_node/Velosity', TwistStamped, queue_size=10)
        self.pub_state =
        rospy.Publisher('/copter_node/State', State, queue_size=10)
        self.pub_setVelocity =
        rospy.Publisher('/mavros/setpoint_velocity/cmd_vel', TwistStamped, queue_size=10)
        self.pub_setPosition =
        rospy.Publisher('/mavros/setpoint_position/local', PoseStamped, queue_size=10)
        self.pub_setHome =
        rospy.Publisher('/mavros/home_position/set', HomePosition, queue_size=10)
        #subscriptions
        rospy.Subscriber('/mavros/local_position/pose', PoseStamped,
        self.localPosition_callback)
        rospy.Subscriber('/mavros/global_position/global', NavSatFix,
        self.globalPosition_callback)
        rospy.Subscriber('/mavros/battery', BatteryState, self.battery_callback)
```

```
rospy.Subscriber('/mavros/local_position/velocity_body', TwistStamped,
self.velocity_callback)
rospy.Subscriber('/mavros/state', State, self.state_callback)
rospy.Subscriber('/copter_node/SetVelocity', TwistStamped,
self.setVelocity_callback)
rospy.Subscriber('/copter_node/SetPosition', Point,
self.setPosition_callback)
#servicios
rospy.Service("/copter_node/Takeoff", takeoff, self.takeoff_function)
rospy.Service("/copter_node/Land", Trigger, self.land_function)
rospy.Service("/copter_node/RTL", Trigger, self.rtl_function)
rospy.Service("/copter_node/Stop", Trigger, self.Stop_function)
rospy.Service("/copter_node/Guided", Trigger, self.Guided_function)
#clientes
self.SetMode_service = rospy.ServiceProxy("/mavros/set_mode", SetMode)
self.Arming_service =
rospy.ServiceProxy("/mavros/cmd/arming", CommandBool)
self.takeoff_service =
rospy.ServiceProxy("/mavros/cmd/takeoff", CommandTOL)
self.mav_frame_velocity_service =
rospy.ServiceProxy("/mavros/setpoint_velocity/mav_frame", SetMavFrame)

self.mav_frame_velocity_service(8)
rospy.loginfo("copter_node iniciado")
self.bucle()

def takeoff_function(self, req):
self.Send_Velocity = False
if self.change_Mode("guided"): #cambio a modo guiado
Armado = self.Arming_service(True) #Armado de los motores
if Armado.success:
Despegue = self.takeoff_service(0,0,0,0, req.altitude) #despegue
if Despegue.success:
self.reach_Point = True
self.Goal_Point =
Point(self.localPosition.x, self.localPosition.y, req.altitude)
return takeoffResponse(True, "Despegue a " + str(req.altitude)
+" m iniciado con éxito")
return takeoffResponse(False, "Error al iniciar el despegue")
else:
return takeoffResponse(False, "Error en el armado de los motores")
else:
return takeoffResponse(False, "Error, no se ha podido cambiar el modo
de vuelo a guiado")

def rtl_function(self, req):
if not self.not_initialPoint:
self.pub_setHome.publish(self.initialPoint)
if self.change_Mode("rtl"):
self.Send_Velocity = False
self.reach_Point = False
return TriggerResponse(True, "Iniciada secuencia de vuelta al punto de
despegue")
return TriggerResponse(False, "Error")

def land_function(self, req):
if self.change_Mode("land"):
self.Send_Velocity = False
self.reach_Point = False
return TriggerResponse(True, "Iniciada secuencia de aterrizaje")
return TriggerResponse(False, "Error")
```

```
def Stop_function(self, req):
    if self.change_Mode("brake"):
        self.Send_Velocity = False
        self.reach_Point = False
        return TriggerResponse(True, "Deteniendo UAV")
    return TriggerResponse(False, "Error")

def Guided_function(self, req):
    if self.change_Mode("guided"):
        return TriggerResponse(True, "Modo guiado activado")
    return TriggerResponse(False, "Error")

#filtro
def localPosition_callback(self, msg):
    self.localPosition = msg.pose.position
    self.pub_localposition.publish(msg)

def globalPosition_callback(self, msg):
    if self.not_inicialPoint:
        #registro del punto inicial
        self.initialPoint = HomePosition()
        self.initialPoint.geo.latitude = msg.latitude
        self.initialPoint.geo.longitude = msg.longitude
        self.initialPoint.geo.altitude = msg.altitude - 19.4
        self.not_inicialPoint = False
    self.pub_globalposition.publish(msg)

def battery_callback(self, msg):
    self.pub_battery.publish(msg)

def velocity_callback(self, msg):
    self.velocity = msg

def state_callback(self, msg):
    self.pub_state.publish(msg)

def setVelocity_callback(self, msg):
    self.Send_Velocity = True
    self.reach_Point = False
    self.Vel_msg = msg
    rospy.loginfo("consigna de velocidad: " + str(msg.twist))

def setPosition_callback(self, msg):
    if not self.Send_Velocity:
        if not self.not_inicialPoint:
            self.pub_setHome.publish(self.initialPoint)
        if self.change_Mode("guided"):
            self.reach_Point = True
            self.Goal_Point = msg
            position_msg = PoseStamped()
            position_msg.pose.position = msg
            yaw = atan2(msg.y - self.localPosition.y, msg.x -
self.localPosition.x)
            Q = quaternion_from_euler(0, 0, yaw)
            position_msg.pose.orientation.x = Q[0]
            position_msg.pose.orientation.y = Q[1]
            position_msg.pose.orientation.z = Q[2]
            position_msg.pose.orientation.w = Q[3]
            self.pub_setPosition.publish(position_msg)
            rospy.loginfo("Dirigiendose al punto: " + str(msg))
        else:
```

```
rospy.loginfo("Error, no se ha podido cambiar el modo de vuelo a guiado")

def bucle(self):
    while not rospy.is_shutdown():
        if self.Send_Velocity:
            self.pub_setVelocity.publish(self.Vel_msg)
        elif self.reach_Point and (self.distance(self.localPosition,self.Goal_Point) < 0.4):
            self.reach_Point = False
            self.change_Mode("brake")
            self.pub_velocity.publish(self.velocity)
            self.rate.sleep()

def change_Mode(self,Mode):
    try:
        if self.SetMode_service(0,Mode).mode_sent:
            return True
        return False
    except rospy.ServiceException as error:
        rospy.loginfo("Ha habido un error: " + str(error))

def distance(self,a,b):
    D = sqrt((b.x - a.x)**2 + (b.y - a.y)**2 + (b.z - a.z)**2)
    return D

if __name__ == "__main__":
    Copter_node()
```

A.2 mission_node.py

```
#!/usr/bin/env python

import rospy
import time
from tfg_alvaro.srv import*
from std_srvs.srv import Trigger, TriggerResponse
from mavros_msgs.srv import WaypointPush, WaypointPushResponse, WaypointPushRequest
from mavros_msgs.srv import WaypointClear, WaypointClearResponse, WaypointClearRequest
from mavros_msgs.srv import SetMode, SetModeResponse, SetModeRequest
from mavros_msgs.msg import Waypoint

class Mission_node():
    def __init__(self):
        rospy.init_node("mission_node")
        #service
        rospy.Service("/mission_node/upload",fileMission,self.upload_function)
        rospy.Service("/mission_node/startMission", Trigger,self.start_function)
        #client
        self.push_service = rospy.ServiceProxy("mavros/mission/push", WaypointPush)
        self.clear_service = rospy.ServiceProxy("mavros/mission/clear", WaypointClear)
        self.takeoff_service = rospy.ServiceProxy("copter_node/Takeoff", takeoff)
        self.SetMode_service = rospy.ServiceProxy("/mavros/set_mode",SetMode)
        rospy.spin()

    def upload_function(self,req):
```

```
mission,error = self.read_file(req.filePath)
if mission == None:
    rospy.loginfo(error)
    return fileMissionResponse(False,error)
try:
    if self.clear_service().success: #borramos mission actual
        if self.push_service(0,mission).success: #cargar nueva mision
            rospy.loginfo("Mission uploaded")
            return fileMissionResponse(True,"Mission uploaded")
            rospy.loginfo("Error: no se pudo cargar la mision correctamente")
            return fileMissionResponse(False,"Error: no se pudo cargar la
mision correctamente")
        except rospy.ServiceException as error:
            rospy.loginfo("Error: " + str(error))

def start_function(self,req):
    self.takeoff_service(10)
    time.sleep(15)
    try:
        if self.SetMode_service(0,'auto').mode_sent:
            rospy.loginfo("Mission started")
            return TriggerResponse(True,"Mission started")
            return TriggerResponse(False,"Error: no se puede iniciar la mision")
        except rospy.ServiceException as error:
            rospy.loginfo("Error: " + str(error))

def read_file(self,file):
    try:
        with open(file) as f:
            Lines = f.readlines()
            data = []
            for line in Lines:
                if line[0] == '#':
                    continue
                num = ''
                waypoint = []
                for i in range(0,len(line)):
                    if line[i] != '\t' and line[i] != '\n':
                        num += line[i]
                    else:
                        waypoint.append(float(num))
                        num = ''
                data.append(waypoint)
            #conversion de los datos leidos al tipo Waypoint usado por el nodo
mavros
            mission = []
            #primer comando lo ignora mavros
            wp = Waypoint()
            wp.frame = 3
            wp.command = 16
            wp.is_current = False
            wp.autocontinue = True
            mission.append(wp)
            #obtencion de cada waypoint
            for waypoint in data:
                if waypoint[3] < 0 or not(waypoint[4] > 0):
                    return None, "Error: Algun campo esta incorrecto, revise el
archivo"
                wp = Waypoint()
                wp.is_current = False
                wp.autocontinue = True
                #cambio de velocidad
```

```
wp.frame = 2
wp.command = 178
wp.param1 = 1 # tipo de velocidad (velocidad en tierra)
wp.param2 = waypoint[4] # velocidad
wp.param3 = -1 # cambio velocidad de los motores en porcentaje(-
1 no cambia)

mission.append(wp)
#dijigirse a un waypoint
wp = Waypoint()
wp.is_current = False
wp.autocontinue = True
wp.frame = 3
wp.command = 16
wp.x_lat = waypoint[0] # latitud
wp.y_long = waypoint[1] # longitud
wp.z_alt = waypoint[2] # altura relativa al punto base
wp.param1 = waypoint[3] # delay
mission.append(wp)
#vuelta a tierra (RTL)
wp = Waypoint()
wp.frame = 3
wp.command = 20
wp.is_current = False
wp.autocontinue = True
mission.append(wp)
rospy.loginfo(mission)
return mission, ""
except:
    return None, "Error: la ruta introducida es incorrecta o algun campo
esta incompleto"

if __name__ == "__main__":
    Mission_node()
```

A.3 Prueba_wp.py

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String
from sensor_msgs.msg import NavSatFix
from mavros_msgs.msg import *
from mavros_msgs.srv import *
import math

def Prueba_wp():
    rospy.init_node('Prueba_wp', anonymous=True)
    push_service = rospy.ServiceProxy('mavros/mission/push', WaypointPush)
    clear_service = rospy.ServiceProxy('mavros/mission/clear', WaypointClear)

    try:
        clear_service()
    except rospy.ServiceException as e:
        rospy.loginfo("Error: " + str(e))

    w1 = []

    wp = Waypoint()
    wp.frame = 3
    wp.command = 22 # takeoff
```



```
wp.is_current = False
wp.autocontinue = True
wp.param1 = 0
wp.param2 = 0
wp.param3 = 0
wp.param4 = 0
wp.x_lat = -35.362395
wp.y_long = 149.165232
wp.z_alt = 10
wl.append(wp)

wp = Waypoint()
wp.frame = 2
wp.command = 178 # cambiar velocidad
wp.is_current = False
wp.autocontinue = True
wp.param1 = 1 #tipo de velocidad(1 con respecto a tierra)
wp.param2 = 4 #velocidad
wp.param3 = -1 #giro motores (-1 no cambio)
wp.param4 = 0
wp.x_lat = 0
wp.y_long = 0
wp.z_alt = 0
wl.append(wp)

wp = Waypoint()
wp.frame = 3
wp.command = 16 #Navigate to waypoint.
wp.is_current = False
wp.autocontinue = True
wp.param1 = 0 # delay
wp.param2 = 0 # radio de aceptacion
wp.param3 = 0 # radio de paso
wp.param4 = 0 # Yaw
wp.x_lat = -35.362395
wp.y_long = 149.165232
wp.z_alt = 10
wl.append(wp)

wp = Waypoint()
wp.frame = 2
wp.command = 20 #RTL
wp.is_current = False
wp.autocontinue = True
wp.param1 = 0 #todos los parametros a cero
wp.param2 = 0
wp.param3 = 0
wp.param4 = 0
wp.x_lat = 0
wp.y_long = 0
wp.z_alt = 0
wl.append(wp)

print(wl)
push_service(start_index=0, waypoints=wl)

if __name__ == '__main__':
    Prueba_wp()
```

A.4 Modelado.py

```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import TwistStamped
from rospy.impl.tcpros_service import ServiceProxy
from std_srvs.srv import Trigger, TriggerResponse, TriggerRequest
from tfg_alvaro.srv import*
import time

def main():
    rospy.init_node('Modelado', anonymous=True)
    pub_vel =
rospy.Publisher('/copter_node/SetVelocity', TwistStamped, queue_size=10) #Topic
envio de velocidades
    takeoff_service = rospy.ServiceProxy('/copter_node/Takeoff', takeoff)
#servicio despegue
    Guidedmode_service = rospy.ServiceProxy('/copter_node/Guided', Trigger)
#servicio modo guiado
    Vellinear = 1 #m/s
    Velangular = 1 #rad/s
    takeoff_service(20) #despegue 20 metros
    time.sleep(45)
    Guidedmode_service() #activamos el modo de control de velocidad
    #EJE X
    vel = TwistStamped()
    pub_vel.publish(vel)
    time.sleep(5)
    vel.twist.linear.x = Vellinear
    pub_vel.publish(vel)
    rospy.loginfo("Primer pulso en X")
    time.sleep(30)
    vel.twist.linear.x = 0
    pub_vel.publish(vel)
    time.sleep(30)
    vel.twist.linear.x = Vellinear
    pub_vel.publish(vel)
    rospy.loginfo("Segundo pulso en X")
    time.sleep(30)
    vel.twist.linear.x = 0
    pub_vel.publish(vel)
    time.sleep(30)
    vel.twist.linear.x = Vellinear
    pub_vel.publish(vel)
    rospy.loginfo("Tercer pulso en X")
    time.sleep(30)
    vel.twist.linear.x = 0
    pub_vel.publish(vel)
    time.sleep(30)
    #Eje Y
    vel.twist.linear.y = Vellinear
    pub_vel.publish(vel)
    rospy.loginfo("Primer pulso en Y")
    time.sleep(30)
    vel.twist.linear.y = 0
    pub_vel.publish(vel)
    time.sleep(30)
    vel.twist.linear.y = Vellinear
```

```
pub_vel.publish(vel)
rospy.loginfo("Segundo pulso en Y")
time.sleep(30)
vel.twist.linear.y = 0
pub_vel.publish(vel)
time.sleep(30)
vel.twist.linear.y = Vellinear
pub_vel.publish(vel)
rospy.loginfo("Tercer pulso en Y")
time.sleep(30)
vel.twist.linear.y = 0
pub_vel.publish(vel)
time.sleep(30)
#Eje Z
vel.twist.linear.z = Vellinear
pub_vel.publish(vel)
rospy.loginfo("Primer pulso en Z")
time.sleep(30)
vel.twist.linear.z = 0
pub_vel.publish(vel)
time.sleep(30)
vel.twist.linear.z = Vellinear
pub_vel.publish(vel)
rospy.loginfo("Segundo pulso en Z")
time.sleep(30)
vel.twist.linear.z = 0
pub_vel.publish(vel)
time.sleep(30)
vel.twist.linear.z = Vellinear
pub_vel.publish(vel)
rospy.loginfo("Tercer pulso en Z")
time.sleep(30)
vel.twist.linear.z = 0
pub_vel.publish(vel)
time.sleep(30)
#Giro eje Z
vel.twist.angular.z = Velangular
pub_vel.publish(vel)
rospy.loginfo("Primer pulso giro")
time.sleep(30)
vel.twist.angular.z = 0
pub_vel.publish(vel)
time.sleep(30)
vel.twist.angular.z = Velangular
pub_vel.publish(vel)
rospy.loginfo("Segundo pulso giro")
time.sleep(30)
vel.twist.angular.z = 0
pub_vel.publish(vel)
time.sleep(30)
vel.twist.angular.z = Velangular
pub_vel.publish(vel)
rospy.loginfo("Tercer pulso giro")
time.sleep(30)
vel.twist.angular.z = 0
pub_vel.publish(vel)
time.sleep(30)

if __name__ == '__main__':
    main()
```

A.5 medidas.m

```
dif = abs(length(out.time) - length(in.time));
out = out(dif + 1:end,:);
%Eje X
VinX = [in.fieldtwistlinearx in.time];
VoutX = [out.fieldtwistlinearx out.time];
%Eje Y
VinY = [in.fieldtwistlineary in.time];
VoutY = [out.fieldtwistlineary out.time];
%Eje Z
VinZ = [in.fieldtwistlinearz in.time];
VoutZ = [out.fieldtwistlinearz out.time];
%Giro
VinYaw = [in.fieldtwistangularz in.time];
VoutYaw = [out.fieldtwistangularz out.time];
%Dibujo
figure;
subplot 221; title('Eje X'); hold on; xlabel('Tiempo'); ylabel('Velocidad')
plot(VinX(:,2),VinX(:,1))
plot(VoutX(:,2),VoutX(:,1))
subplot 222; title('Eje Y'); hold on; xlabel('Tiempo'); ylabel('Velocidad')
plot(VinY(:,2),VinY(:,1))
plot(VoutY(:,2),VoutY(:,1))
subplot 223; title('Eje Z'); hold on; xlabel('Tiempo'); ylabel('Velocidad')
plot(VinZ(:,2),VinZ(:,1))
plot(VoutZ(:,2),VoutZ(:,1))
subplot 224; title('Giro'); hold on; xlabel('Tiempo'); ylabel('Velocidad angular')
plot(VinYaw(:,2),VinYaw(:,1))
plot(VoutYaw(:,2),VoutYaw(:,1))
```

B. Modificación de CMakeList

Para el uso de los nodos que comento en este proyecto, se ha tenido que realizar una serie de operaciones. Primero, en el fichero “CMakeList.txt” del paquete, se ha añadido la lista completa de dependencias que utilizan los nodos tal y como se observa en el siguiente fragmento:

```
find_package(catkin REQUIRED COMPONENTS
  rospy
  geometry_msgs
  sensor_msgs
  std_msgs
  std_srvs
  mavros_msgs
  message_generation
)
```

De estas dependencias destaca *message_generation* el cual permite utilizar mensajes y servicios propios. Luego se ha añadido tanto los servicios que se han creado como las dependencias que utilizan, tal y como se observa en la siguiente sección del código:

```
## Generate services in the 'srv' folder
add_service_files(
  FILES
  takeoff.srv
  fileMission.srv
)

## Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Para finalmente poder utilizar estos servicios, se ha descomentado las siguientes dos líneas del archivo package.xml:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

Volviendo al fichero “CMakeList.txt”, se añadió cada uno de los nodos del paquete de la siguiente forma:

```
catkin_install_python(PROGRAMS
  scripts/copter_node.py
  scripts/Modelado.py
  scripts/Prueba_wp.py
  scripts/mission_node.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

A continuación, se compiló el paquete introduciendo la siguiente secuencia de comandos en una terminal. Python es un lenguaje interpretado, es decir, que los códigos no se compilan, sino que un intérprete traduce y ejecuta una a una cada línea del programa. Por tanto, una vez compilado el paquete no hace falta volver a realizar esta operación siempre y cuando no se vuelva a modificar el fichero “CMakeList.txt”.

```
cd ~/catkin_ws  
catkin_make
```

Por último, es necesario conceder a los nodos escritos en Python permisos de ejecución. Para ello se ha introducido el siguiente comando en una terminal. Esta orden solo hace falta introducirla una vez por cada script.

```
chmod +x <script de python>
```